High Performance Journal of Development

High Performance Fortran Forum

CRPC-TR93300
May, 1993

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

High Performance Fortran
Journal of Development

High Performance Fortran Forum

May 3, 1993
Version 1.0

The High Performance Fortran Forum (HPFF), with participation from over 40 organizations, met from March 1992 to March 1993 to discuss and define a set of extensions to Fortran called High Performance Fortran (HPF). Our goal was to address the problems of writing data parallel programs for architectures where the distribution of data impacts performance. While we hope that the HPF extensions will become widely available, HPFF is not sanctioned or supported by any official standards organization.

This is the Journal of Development of the High Performance Fortran Forum. This document contains features proposed for HPF but not included in Version 1.0. It is hoped that the Journal of Development will preserve topics of interest for consideration in a possible follow-on to HPF. This copy of the draft was processed by LaTeX on May 24, 1993.

HPFF encourages requests for interpretation of this document, and comments on the language defined here. We will give our best effort to answering interpretation questions, and general comments will be considered in future HPFF language specifications.

Please send interpretation requests to `hpff-interpret@cs.rice.edu`. Your request is archived and forwarded to a group of HPFF committee members who attempt to respond to it.

Please send comments on the HPF language to `hpff-comments@cs.rice.edu`. Your comment is archived. Periodically, the archives are sent to HPFF committee members for their perusal. Where appropriate, comments are forwarded to the `hpff-interpret` list. HPFF invites comments on the technical content of HPF, as well as on the editorial presentation in the document.

The text of interpretation requests and comments on the language specification become the property of Rice University.

This material appeared in *Scientific Programming*, vol. 2, no. 1, June 1993.

Contents

Section 1

Overview

This document is the Journal of Development for the High Performance Fortran (HPF) language. It contains proposals considered for High Performance Fortran but not included for any of several reasons, including:

- lack of time to complete within the one year time frame of HPF, or
- lack of consensus on the syntax, semantics, or pragmatics of the proposed feature.

In most cases it was felt that the features in this Journal of Development represented capabilities that should be in a language for high performance computing. The High Performance Fortran Forum decided that these proposals should be preserved as input to a potential future HPF-II process.

Information on the goals of HPF, its specific features, and credits to the members of the High Performance Fortran Forum are found in the companion document *High Performance Fortran Language Specification.*

Section 2

Input/Output

High Performance Fortran has exactly the same Input/Output facilities that are available in Fortran 90.

One of the High Performance Fortran Forum working groups was dedicated to I/O extensions for parallel machines. The objective was to define a set of extensions to standard Fortran 90 I/O which would provide high I/O performance on a massively parallel computer running HPF.

Three proposals in this spirit were offered. The basic idea of these proposals is outlined here and the proposals themselves appear in Section 13.

The HPFF voted not to include I/O extensions in the first version of the language. Arguments for this position include:

- I/O systems on parallel computers from different vendors are too architecturally different for there to be a useful abstraction on which the language model can build. For example, consider the difference between a machine with disks on each node, compared with one which has a high bandwidth disk system connected to the communication network, and thus globally accessible.
- Fortran I/O is already highly expressive, some would say too expressive.
- The HPF compiler must already 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.

Moreover the current lack of extensions does **not** limit features that may be added by system vendors. In particular:

- Vendors are allowed to implement any I/O extensions to the language they may wish. Indeed this would be impossible to prevent. There are simply no special I/O mechanisms mandated by HPF.
- The HPF run-time system may use whatever facilities the operating system provides for accessing "high performance" files, though the HPF language contains no I/O extensions that specifically describe such access. For example, the HPF system is entirely free to place a status='SCRATCH' file in the highest performance file system

it likes, and distribute it as is appropriate for the machine, but it is not up to the user to say all this.

The proposals made to the I/O subgroup were based on the following observations:

- A massively parallel machine needs massively parallel I/O;
- Efficient programs must avoid sequential bottlenecks from processors to file systems; and
- Fortran specifies that a file appears in element storage order; this conflicts with striped files (for example, an array distributed by rows may be written to a file striped by columns).

The proposals were that HPF should provide explicit control to obtain high performance I/O. In essence the three proposals were:

1. On a write, give a hint about how the data will be read.

```
!HPF$ DISTRIBUTE (CYCLIC) :: a
!HPF$ IO_DISTRIBUTE * :: a
WRITE a, b, c
```

   When an array is written, it can be easily read back in the given distribution. The annotation can be associated with either the declaration or the write itself; in the first case it applies to all writes of the array, while in the second it only applies to the one statement. The intent is that metadata is kept in the file system to record the "right" data layout. The advantages of this proposal include notation and efficiency.

2. Give hints about the physical layout (number of spins, record length, striping function, etc.) of the file when it is opened.

   This uses the HPF array mapping mechanisms. (A file is a 1-dimensional array of records.) The syntax needs a "name" for the file "template"; the proposal is to use FILEMAP. The programmer can align/distribute FILEMAP (on I/O nodes), associate FILEMAP with a file on OPEN, etc. There are no changes in semantics or file system.

3. Introduce parallel read/write operations that are not necessarily compatible with sequential ones.

```
PWRITE a
PREAD a
```

   Data can be read back only into arrays of the same shape and mapping. Data written by PWRITE must be read by PREAD. This solution does not need metadata in the file system or changes in the file system but is incompatible with the standard READ and WRITE.

Section 3

VIEW Directive

The `VIEW` attribute provides a mechanism 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.")

J301 *view-directive* **is** `VIEW` *processor-view view-attribute-stuff*

J302 *view-attribute-stuff* **is** `OF` *processor-viewed*

J303 *processor-view* **is** *processor-name permutation*

J304 *processor-view-entity* **is** *processor-name* [`(` *array-spec* `)`] [*permutation*]

J305 *processor-viewed* **is** *processor-name* [*permutation*]

J306 *permutation* **is** *array-constructor*

Constraint: A permutation, if present, must be a constant integer array, syntactically expressed as an array constructor, whose elements are a permutation of the values $(1, 2, \ldots, n)$, where n is the rank of the associated processor-name. If it is omitted and the processor-name names a non-scalar processors arrangement, it is as if the identity permutation `(/ 1, 2, ..., `*n*` /)` had been specified.

Constraint: The `VIEW` directive may appear only in a *declaration-part* of a program.

Note that the possibility of a `VIEW` directive of the form

`!HPF$ VIEW` *view-attribute-stuff* `::` *processor-view-list*

is covered by syntax rule H301 for a *combined-directive* in the High Performance Fortran Language Specification.

The `VIEW` directive relates each of a list of processors arrangement names, each with an optional permutation array, to one specific other processors arrangement, which may also have a permutation array. The relationship between a view A and a viewed arrangement B is established as follows. If A is scalar, B must be scalar, and the permutation must be absent or empty; in this case A and B designate the same abstract processor. If A is non-scalar, B must be non-scalar, with the same size as A but not necessarily the same shape or rank. Let P be the permutation array for A and Q be the permutation array for B; let M be the rank of A (and thus the length of P) and let N be the rank of B (and

thus the length of Q). Permute the dimensions of A, yielding a processor array A' such that dimension J of A' corresponds to dimension $P(J)$ of A for $1 \leq J \leq M$. Similarly permute the dimensions of B, yielding a processor array B' such that dimension K of B' corresponds to dimension $Q(K)$ of B for $1 \leq K \leq N$. The permuted processor arrays A' and B' are then "equivalenced" using Fortran's usual column-major order.

```
      PARAMETER (A = (/2,1/))
      DIMENSION A(2)

!HPF$ PROCESSORS P(10,10), Q(100), R(100)
!HPF$ PROCESSORS S(10,10), T(100)

!HPF$ VIEW OF P :: Q
!HPF$ VIEW OF P(/2,1/) :: R

!HPF$ VIEW S(/A/) OF T
```

In the code fragment above, the processor arrays `P`, `Q` and `R` designate the same set of 100 abstract processors in different ways. Because `P` does not appear as a *processor-view*, it designates the same two-dimensional arrangement as any other 10×10 processor arrangement with no `VIEW` attribute.

The first `VIEW` directive specifies that `Q` names the same set of processors as `P`, in such a way that, for all `I` in the range `1:10` and all `J` in the range `1:10`, `P(I,J)` and `Q((J-1)*10+I)` designate the same processor.

The second `VIEW` directive specifies that `R` names the same set of processors as `P` after permuting the dimensions of the latter. Thus, in this case `P(I,J)` and `R((I-1)*10+J)` designate the same processor.

The third `VIEW` directive specifies that `S`, taken in row-major order, provides a view of the one-dimensional processor arrangement `T`. A named parameter constant `A` is used to specify the permutation.

The `VIEW` attribute may appear in a combined-directive such as

```
!HPF$ PROCESSORS WILL_YOU_STILL_NEED_ME__WHEN_IM(64)
!HPF$ PROCESSORS, VIEW OF WILL_YOU_STILL_NEED_ME__WHEN_IM,   &
!HPF$      DIMENSION(4,4,4) :: RUBIKS_REVENGE
!HPF$ PROCESSORS,VIEW OF RUBIKS_REVENGE(/2,3,1/) ::     &
!HPF$            CHESSBOARD(8,8)(/2,1/)
```

For all `I` in the range `1:8` and all `J` in the range `1:8`, `CHESSBOARD(I,J)` means the same processor as `RUBIKS_REVENGE((I-1)/2+1, IAND(J-1,3)+1, 2*IAND(I-1,1)+(J-1)/4+1)`.

The `VIEW` directive requires some modifications to other syntax rules:

J307 *combined-directive* **is** *combined-attribute-list* `::` *hpf-entity-decl-list*

J308 *combined-attribute* **is** ...
or `VIEW` *view-attribute-stuff*

J309 *hpf-entity-decl* **is** *entity-decl*
or *processor-view-entity*

Also, the remark about procesor arrangements:

> If two processor arrangements have the same shape, then corresponding elements of the two arrangements are understood to refer to the same abstract processor.

should be modified to read

> If two processor arrangements have the same shape and neither has the `VIEW` attribute, then corresponding elements of the two arrangements are understood to refer to the same abstract processor. The use of the `VIEW` attribute may also cause an element of one processor arrangement to refer to the same abstract processor as an element of some other arrangement.

Furthermore:

> Whenever the subprogram is called, the processor arrangement is always locally defined in the same way, with identical lower bounds and identical upper bounds.

should be modified to read

> Whenever the subprogram is called, the processor arrangement is always locally defined in the same way, with identical lower bounds, identical upper bounds, and identical view attribute information (if any).

Some concerns about the `VIEW` directive as specified above:

Right now some extremely complicated mappings can be achieved, particularly by chaining `VIEW` directives (currently there is no restriction against chaining).

By chaining `VIEW` directives, it is possible to rearrange all the prime factors of all the dimensions of the original *view-target* into any order whatsoever.

Even without chaining, one can construct an 8×27 view of a $6 \times 6 \times 6$ array. Did we really want this generality?

Here are four alternative proposals to consider.

1. Eliminate the VIEW directive.
2. Forbid chained views; that is, if `Q` is a view of `R`, then `P` may not be a view of `Q`, but only of `R` directly.
3. If `P` is a view of `Q`, then there must be a set of arrows from the permuted dimensions of `P`, laid out in order in a line, to the permuted dimensions of `Q`, laid out in order in a line, such that:
   (a) every dimension is at one end of at least one arrow;
   (b) for every arrow, that arrow either shares the dimension at its tail with no other arrow or shares the dimension at its head with no other arrow;
   (c) a dimension shared by several arrows must equal the product of the dimensions at the other ends of the arrows; and
   (d) no arrows cross.
4. Both 2 and 3 together.

Other possibilities considered by the committee included:

- Restricting a view to be one-dimensional (and prohibiting chaining, as the value of chaining is trivial given this restriction).
- Restricting a view to be of lower rank than the viewed arrangement.
- Restricting a view to be of lower rank than the viewed arrangement and prohibiting chaining.

Section 4

User-Specified Default Distributions

The HPF specification states that if the *dist-format-clause* is omitted from the attributed form of a `DISTRIBUTE` or `REDISTRIBUTE` directive, then the language processor may make an arbitrary choice of distribution formats for each template or array.

It would be convenient for the user to be able to specify defaults to be applied in this situation, rather than leaving the choice entirely to the language processor. Think of it as "`IMPLICIT`" distributions.

One specific proposal is to have a separate default for each possible array rank; given the current Fortran 90 limit of 7 on array rank, there would be seven defaults. A default for a scoping unit could be specified by a `DISTRIBUTE` directive that mentions no `distributee`. For example:

```
!HPF$ PROCESSORS ALLPROCS(512),XY_PROCS(16,32)
!HPF$ DISTRIBUTE (BLOCK) ONTO ALLPROCS
!HPF$ DISTRIBUTE (CYCLIC,CYCLIC) ONTO XY_PROCS
!HPF$ DISTRIBUTE (CYCLIC,CYCLIC,*) ONTO XY_PROCS
!HPF$ DISTRIBUTE (*,CYCLIC,CYCLIC,*) ONTO XY_PROCS
```

would mean that any one-dimensional array or template in the scoping unit for which no explicit alignment or distribution is specified would be distributed `(BLOCK)` onto `ALLPROCS`; any two-dimensional array or template not explicitly mapped would be distributed `(CYCLIC, CYCLIC)` onto `XY_PROCS`; and so on.

Section 5

Partially Specified Distribution Formats

A programmer might wish to dictate the distribution format for some axes of a template while leaving to the language processor the choice of distribution format for other axes. The specific suggestion would allow a *dist-format* to be empty. For example,

```
!HPF$ DISTRIBUTE A(BLOCK,,*,) ONTO SCORPROCS
```

would dictate `BLOCK` distribution for the first axis and on-processor distribution for the third axis, leaving the choice of distribution formats for the second and fourth axes to the compiler.

An alternate suggestion was that, as there was no precedent in Fortran 90 for such syntactically empty list items, some explicit token such as `?` should be chosen.

Section 6

Unanchored Distribution Formats

Consider the following program fragment:

```
      REAL A(100),B(100)
!HPF$ DISTRIBUTE (BLOCK(10)) ONTO P :: A,B
      A(1:90) = B(11:100)
```

Question: is this understood to imply that an HPF compiler that purports to obey the directives should not map `A(1)` and `B(11)` to the same processor, thereby avoiding communication overhead?

Now ask the same question about this version, which omits `ONTO P`:

```
      REAL A(100),B(100)
!HPF$ DISTRIBUTE (BLOCK(10)) :: A,B
      A(1:90) = B(11:100)
```

Some have suggested that the latter case, at least, dictates a blocking factor of 10 but says nothing about relative alignment, and therefore it would be within the spirit of HPF for a compiler to choose an offset alignment such that `A(1:10)` and `B(11:20)` are mapped to the same processor, etc.

Section 7

Nested WHERE statements

Briefly put, the less WHERE is like IF, the more difficult it is to translate existing serial codes into array notation. Such codes tend to have the general structure of one or more DO loops iterating over array indices and surrounding a body of code to be applied to array elements. Conversion to array notation frequently involves simply deleting the DO loops and changing array element references to array sections or whole array references. If the loop body contains logical IF statements, these are easily converted to WHERE statements. The same is true for translating IF-THEN constructs to WHERE constructs, except in two cases. If the IF constructs are nested (or contain IF statements), or if ELSE IF is used, then conversion suddenly becomes disproportionately complex, requiring the user to create temporary variables or duplicate mask expressions and to use explicit .AND. operators to simulate the effects of nesting.

Users also find it confusing that ELSEWHERE is syntactically and semantically analogous to ELSE rather than to ELSE IF.

We therefore propose that the syntax of WHERE constructs be extended and changed to have the form

J701 *where-construct* **is** *where-construct-stmt*
[*where-body-construct*]...
[*elsewhere-stmt*
[*where-body-construct*]...]...
[*where-else-stmt*
[*where-body-construct*]...]
end-where-stmt

J702 *where-construct-stmt* **is** `WHERE (` *mask-expr* `)`

J703 *elsewhere-stmt* **is** `ELSE WHERE (` *mask-expr* `)`

J704 *where-else-stmt* **is** `ELSE WHERE`

J705 *end-where-stmt* **is** `END WHERE`

J706 *mask-expr* **is** *logical-expr*

J707 *where-body-construct* **is** *assignment-stmt*
or *where-stmt*
or *where-construct*

Constraint: In each *assignment-stmt*, the *mask-expr* and the variable being defined must be

arrays of the same shape. If a *where-construct* contains a *where-stmt*, an *elsewhere-stmt*, or another *where-construct*, then the two *mask-expr*'s must be arrays of the same shape.

The meaning of such statements may be understood by rewrite rules. First one may eliminate all occurrences of ELSE WHERE:

```
WHERE (m1)                          WHERE (m1)
  xxx                                 xxx
ELSE WHERE (m2)       becomes       ELSE
  yyy                                 WHERE (m2)
END WHERE                               yyy
                                      END WHERE
                                    END WHERE
```

where xxx and yyy represent any sequences of statements, so long as the original WHERE, ELSE WHERE, and END WHERE match, and the ELSE WHERE is the first ELSE WHERE of the construct (that is, yyy may include additional ELSE WHERE or ELSE statements of the construct). Next one eliminates ELSE:

```
WHERE (m)                           temp = m
  xxx                               WHERE (temp)
ELSE                    becomes        xxx
  yyy                               END WHERE
END WHERE                           WHERE (.NOT. temp)
                                      yyy
                                    END WHERE
```

Finally one eliminates nested WHERE constructs:

```
WHERE (m1)                          temp = m1
  xxx                               WHERE (temp)
  WHERE (m2)                          xxx
    yyy                 becomes     END WHERE
  END WHERE                         WHERE (temp .AND. (m2))
  zzz                                 yyy
END WHERE                           END WHERE
                                      WHERE (temp)
                                        zzz
                                      END WHERE
```

and similarly for nested WHERE statements.

The effects of these rules will surely be a familiar or obvious possibility to all the members of the committee; I enumerate them explicitly here only so that there can be no doubt as to the meaning I intend to support.

Such rewriting rules are simple for a compiler to apply, or the code may easily be compiled even more directly. But such transformations are tedious for our users to make by hand and result in code that is unnecessarily clumsy and difficult to maintain.

One might propose to make WHERE and IF even more similar by making two other changes. First, require the noise word THERE to appear in a WHERE and ELSE WHERE statement after the parenthesized mask-expr, in exactly the same way that the noise word THEN must appear in IF and ELSE IF statements. (Read aloud, the results might sound a trifle old-fashioned–"Where knights dare not go, there be dragons!"–but technically would be as grammatically correct English as the results of reading an IF construct aloud.) Second, allow a WHERE construct to be named, and allow the name to appear in ELSE WHERE, ELSE, and END WHERE statements. I do not feel very strongly one way or the other about these no doubt obvious points, but offer them for your consideration lest the possibilities be overlooked.

Now, for compatibility with Fortran 90, HPF should continue to use ELSEWHERE instead of ELSE, but this causes no ambiguity:

```
WHERE(...)
  ...
ELSE WHERE(...)
  ...
ELSEWHERE
  ...
END WHERE
```

is perfectly unambiguous, even when blanks are not significant(fixed source form). Since X3J3 declined to adopt the keyword THERE, it should not be used in HPF either (alas), though it could be allowed optionally.

Section 8

ALLOCATE in FORALL

Proposal: ALLOCATE, DEALLOCATE, and NULLIFY statements may appear in the body of a FORALL.

Rationale: These are just another kind of assignment. They may have a kind of side effect (storage management), but it is a benign side effect (even milder than random number generation).

Example:

```
      TYPE SCREEN
        INTEGER, POINTER :: P(:,:)
      END TYPE SCREEN
      TYPE(SCREEN) :: S(N)
      INTEGER IERR(N)
      ...
!  Lots of arrays with different aspect ratios
      FORALL (J=1:N)  ALLOCATE(S(J)%P(J,N/J),STAT=IERR(J))
      IF(ANY(IERR)) GO TO 99999
```

Section 9

Generalized Data References

Proposal: Delete the constraint in section 6.1.2 of the Fortran 90 standard (page 63, lines 7 and 8):

> Constraint: In a data-ref, there must not be more than one part-ref with nonzero rank. A part-name to the right of a part-ref with nonzero rank must not have the POINTER attribute.

Rationale: Further opportunities for parallelism.
Example:

```
TYPE MONARCH
    INTEGER, POINTER :: P
END TYPE MONARCH
TYPE(MONARCH) :: C(N), W(N)
      ...
! Munch that butterfly
C = C + W * A%P ! Illegal in Fortran 90
```

Section 10

FORALL with INDEPENDENT Directives

We propose that two new directives be added for use within the FORALL construct.

```
!HPF$BEGIN INDEPENDENT
!HPF$END INDEPENDENT
```

The two directives must be used in pair. A sub-block of statements parenthesized in the two directives is called an *asynchronous* sub-block or *independent* sub-block. The statements that are not in an asynchronous sub-block are in *synchronized* sub-blocks or *non-independent* sub-block. The synchronized sub-block is the same as the HPF FORALL construct, and the asynchronous sub-block is the same as the FORALL with the INDEPENDENT directive. Thus, the block FORALL

```
      FORALL (e)
        b1
!HPF$BEGIN INDEPENDENT
        b2
!HPF$END INDEPENDENT
        b3
      END FORALL
```

means the same as

```
      FORALL (e)
        b1
      END FORALL
!HPF$INDEPENDENT
      FORALL (e)
        b2
      END FORALL
      FORALL (e)
        b3
      END FORALL
```

The INDEPENDENT directive indicates to the compiler there is no dependence and consequently, synchronizations are not necessary. It is users' responsibility to ensure there is no dependence between instances in an asynchronous sub-block.

10.1 What Does "No Dependence Between Instances" Mean?

It means that there is no true dependence, anti-dependence, or output dependence between instances. Examples of these dependences are shown below:

1. True dependence:

```
FORALL (i = 1:N)
  x(i) = ...
  ...  = x(i+1)
END FORALL
```

   Notice that dependences in FORALL are different from that in a DO loop. If the above example was a DO loop, that would be an anti-dependence.

2. Anti-dependence:

```
FORALL (i = 1:N)
  ...  = x(i+1)
  x(i) = ...
END FORALL
```

3. Output dependence:

```
FORALL (i = 1:N)
  x(i+1) = ...
  x(i) = ...
END FORALL
```

10.2 Rationale

1. A FORALL with a single asynchronous sub-block is the same as a DO with an INDEPENDENT assertion. A FORALL with no INDEPENDENT directive is the same as a tightly synchronized FORALL. We only need to define one type of parallel construct including both synchronized and asynchronous blocks. Furthermore, combining asynchronous and synchronized FORALLs, we have a loosely synchronized FORALL which is more flexible for many loosely synchronous applications.

2. With INDEPENDENT directives, the user can indicate which block needs not to be synchronized. The INDEPENDENT directives can act as barrier synchronizations.

Section 11

EXECUTE-ON-HOME and LOCAL-ACCESS Directives

The EXECUTE-ON-HOME directive is used to suggest where an iteration of a DO construct or an indexed parallel assignment should be executed. The specified location of computation provides the reference with which the compiler determines which data access of the computation should be local and which data access may be remote. The LOCAL-ACCESS directive further asserts which data accesses are indeed local.

J1101 *execute-on-home-direct* **is** `EXECUTE (`*align-source-list*`) ON_HOME` *align-spec*
[`,` *local-access-directive*]

J1102 *local-access-directive* **is** `LOCAL_ACCESS` *array-name-list*

The EXECUTE-ON-HOME directive must immediately precede the corresponding DO construct, array assignment, FORALL statement, FORALL construct or individual assignment statement in a FORALL construct.

The scope of an EXECUTE-ON-HOME directive is the entire loop body of the following DO construct, or the following array assignment, FORALL statement, FORALL construct or assignment statement in a FORALL construct.

When an EXECUTE-ON-HOME directive is applied to a DO construct, a FORALL statement, a FORALL construct or an assignment statement in a FORALL construct, the *align-source-list* identifies a distinct iteration index or an indexed parallel assignment in the corresponding scope and the *align-spec* identifies a template node. Every iteration index or indexed assignment must be associated with one and only one template node. The EXECUTE-ON-HOME directive states that each iteration or indexed parallel assignment should be executed on the processor where its associated template node is mapped. For any subroutine call within a DO construct, the EXECUTE-ON-HOME directive specifies only the execution location of the caller but not necessarily the execution location of the called subroutine.

When an EXECUTE-ON-HOME directive is applied to an array assignment statement, each *align-source* identifies positions spread along one dimension (:) or a collapsed dimension (*) of the assigned array, and the *align-spec* identifies the associated template or template

section. (Replication, i.e. "*", is not allowed in *align-spec*.) The *align-source-list* must have the same rank as the assigned array. The associated template or template section must have the same size as the assigned array in all uncollapsed dimensions. The EXECUTE-ON-HOME directive states that, for each element in the assigned array, the corresponding evaluation and assignment should be executed on the processor where the corresponding template element of the associated template is mapped. For example,

```
!HPF$ EXECUTE (:,*) ON_HOME T(2:N)
      A(1:N-1,2:N) = B(2:N,1:N-1)
```

A(1,j) = B(2,j-1) is executed on the processor to which T(2) is mapped, where j = 2,..,N.

(*Align-spec* in current HPF is restricted to simple expressions of *align-dummy*. Thus only regular data mapping is supported. If array elements or functions are allowed in *align-spec* for specifying irregular data mapping, the above EXECUTE-ON-HOME directive can also be used to address the corresponding computation location problem.)

EXECUTE-ON-HOME directives can be nested, but only the immediately preceding EXECUTE-ON-HOME directive is effective.

The optional LOCAL-ACCESS directive asserts that all data accesses to the specified *array-name-list* within the scope of the EXECUTE-ON-HOME directive can be handled as local data accesses if the related HPF data mapping directives are honored.

The LOCAL-ACCESS directive can also be used separately from the EXECUTE-ON-HOME directive. When used alone, it applies only to the immediately following statement or construct, and asserts that all specified data accesses are local data accesses provided that the immediately preceding EXECUTE-ON-HOME directive and all related HPF data mapping directives are honored. The assertion overrides any local-access assertions by the preceding EXECUTE-ON-HOME directive. It is an error when a LOCAL-ACCESS directive is not applied inside the scope of some EXECUTE-ON-HOME directive.

INDEPENDENT and EXECUTE-ON-HOME directives can be combined into a single HPF directive when they are applied to the same DO or FORALL construct,

J1103 *combined-assert-direct* **is** *assertion-directive-list*

J1104 *assertion-directive* **is** *independent-directive*
or *execute-on-home-directive*

Example 1

```
      REAL A(N), B(N), C(N)
!HPF$ TEMPLATE T(N)
!HPF$ ALIGN WITH T:: A, B, C
!HPF$ DISTRIBUTE T(CYCLIC(2))

!HPF$ INDEPENDENT, EXECUTE (I) ON_HOME T(2*I), LOCAL_ACCESS A, B, C
      DO I = 1, N/2
      ! we know that P(2*I-1) and P(2*I) is a permutation
```

```
        ! of 2*I-1 and 2*I
          A(P(2*I - 1)) = B(2*I - 1) + C(2*I - 1)
          A(P(2*I)) = B(2*I) + C(2*I)
        END DO
```

Example 2

```
        REAL A(N,N), B(N,N)
  !HPF$ TEMPLATE T(N,N)
  !HPF$ ALIGN WITH T:: A, B
  !HPF$ EXECUTE (I,J) ON_HOME T(I+1,J-1)
        FORALL (I=1:N-1, J=2:N)   A(I,J) = A(I+1,J-1) + B(I+1,J-1)
```

Example 3

```
        REAL A(N,N), B(N,N)
  !HPF$ TEMPLATE T(N,N)
  !HPF$ ALIGN WITH T:: A, B

  !HPF$ EXECUTE (:,:) ON_HOME T(2:N,1:N-1)
        A(1:N-1,2:N) = A(2:N,1:N-1) + B(2:N,1:N-1)
```

Example 4 The original program for this example is due to Michael Wolfe of Oregon Graduate Institute.

This program performs matrix multiplication $C = A \times B$ by a systolic algorithm. Note that without the EXECUTE-ON-HOME and LOCAL_ACCESS directive, the compiler will have a hard time detecting that all A, B and C accesses are actually local.

```
        REAL A(N,N), B(N,N), C(N,N)

        PARAMETER(NOP = NUMBER_OF_PROCESSORS())
  !HPF$ DYNAMIC B
  !HPF$ TEMPLATE T(2*N,N)               ! to allow wrap around mapping
  !HPF$ ALIGN (I,J) WITH T(I,J):: A, C
  !HPF$ ALIGN B(I,J) WITH T(N+I,J)
  !HPF$ DISTRIBUTE T(CYCLIC(N/NOP),*)   ! distributed by row blocks

        IB = N/NOP

        DO IT = 0, NOP-1

        ! rotate B by row-blocks
  !HPF$ REALIGN B(I,J) WITH T(N-IT*IB+I,J)

        ! data parallel loop
```

```
!HPF$ INDEPENDENT
!HPF$ EXECUTE (IP) ON_HOME T(IP*IB+1,1), LOCAL_ACCESS A, B, C

        DO IP = 0, NOP-1
          ITP = MOD( IT+IP, NOP )
          DO I = 1, IB
            DO J = 1, N
              DO K = 1, IB
                C(IP*IB+I,J) = C(IP*IB+I,J) +
     1                         A(IP*IB+I,ITP*IB+K)*B(ITP*IB+K,J)
              ENDDO  ! K
            ENDDO  ! J
          ENDDO  ! I
        ENDDO  ! IP

       ENDDO  ! IT
```

Section 12

Elemental Reference of Pure Procedures

Fortran 90 introduces the concept of "elemental procedures", which are defined for scalar arguments but may also be applied to conforming array-valued arguments. The latter type of reference to an elemental procedure is called an "elemental" reference. Examples are the mathematical intrinsics, e.g. SIN and the intrinsic subroutine MVBITS. However, Fortran 90 restricts elemental reference to a subset of the intrinsic procedures — programmers cannot define their own elemental procedures. We propose that pure procedures may also be referenced elementally, subject to certain additional constraints given below.

12.1 Elemental Reference of Pure Functions

A user-defined pure function may be referenced *elementally*, provided it satisfies the additional constraints that:

1. Its dummy arguments (except procedure dummy arguments) and result are scalar and do not have the POINTER attribute.

2. The length of any character dummy argument or result is independent of argument values (though it may be assumed, or depend on the lengths of other character arguments and/or a character result).

We call user-written pure functions that satisfy these constraints "elemental non-intrinsic functions", and use the term "elemental function" to include both elemental intrinsic and non-intrinsic functions.

The interpretation of an elemental reference of such a function is as follows (adapted from Section 12.4.3 of the Fortran 90 standard):

> A reference to an elemental function is an elemental reference if one or more actual arguments are arrays and all array arguments have the same shape. If any actual argument is a function, its result must have the same shape as that of the corresponding function dummy procedure.
>
> The result of such a reference has the same shape as the array arguments, and the value of each element of the result, if any, is obtained by evaluating the function using the scalar and procedure arguments and the corresponding

elements of the array arguments. The elements of the result may be evaluated in any order.

Example:

```
INTERFACE
  REAL, ELEMENTAL FUNCTION foo (x, y, z, dummy_func)
    REAL, INTENT(IN) :: x, y, z
    INTERFACE        ! interface for 'dummy_func''
      REAL, ELEMENTAL FUNCTION dummy_func (x)
        REAL, INTENT(IN) :: x
      END FUNCTION dummy_func
    END INTERFACE
  END FUNCTION foo
END INTERFACE

REAL a(100), b(100), c(100)

c = foo (a, 0.0, b, sin)
```

12.2 Elemental Reference of Pure Subroutines

A user-defined pure subroutine may be referenced *elementally*, provided it satisfies the additional constraints that:

1. Its dummy arguments (except procedure dummy arguments) are scalar and do not have the POINTER attribute.
2. The length of any character dummy argument is independent of argument values (though it may be assumed, or depend on the lengths of other character arguments).
3. None of the dummy arguments allow alternate return specifiers.

We call user-written pure subroutines that satisfy these constraints "elemental non-intrinsic subroutines", and use the term "elemental subroutine" to include both elemental intrinsic and non-intrinsic subroutines.

The interpretation of an elemental reference of such a subroutine is as follows (adapted from Section 12.4.5 of the Fortran 90 standard):

> A reference to an elemental subroutine is an elemental reference if all actual arguments corresponding to INTENT(OUT) and INTENT(INOUT) dummy arguments are arrays that have the same shape and the remaining actual arguments (except procedure dummy arguments) are conformable with them. If any actual argument is a function, its result must have the same shape as that of the corresponding function dummy procedure.
>
> The values of the elements of the arrays that correspond to INTENT(OUT) and INTENT(INOUT) dummy arguments are the same as if the subroutine were invoked separately, in any order, using the scalar and procedure arguments and corresponding elements of the array arguments.
>
> Example:

```
INTERFACE
  ELEMENTAL SUBROUTINE solve_simul(tol, y, z)
    REAL, INTENT(IN) :: tol
    REAL, INTENT(INOUT) :: y, z
  END SUBROUTINE
END INTERFACE

REAL a(100), b(100), c(100)

CALL solve_simul( 0.1, a, b )
CALL solve_simul( c(10:100:10), a(1:10), b(1:10) )
```

12.3 Constraints

It is perhaps worth outlining the reasons for the extra constraints imposed on pure procedures in order for them to be referenced elementally.

The result of an elemental function or "output" arguments of a subroutine are not allowed to have the POINTER attribute because Fortran 90 does not permit an array of pointers to be referenced. The "input" arguments of an elemental reference are prohibited from having the POINTER attribute for consistency with the output arguments or result.

In an elemental reference, any actual argument that is a function must have a result whose shape agrees with that of the corresponding function dummy procedure. That is, elemental usage does not extend to function arguments, as Fortran 90 does not support the concept of an "array" of functions.

Finally, the length of any character dummy argument or a character dummy result cannot depend on argument *values* (though it can be assumed, or depend on the lengths of other character arguments). This ensures that under elemental reference, all elements of an array argument or result of character type will have the same length, as required by Fortran 90.

Section 13

Parallel I/O

High Performance Fortran is primarily designed to obtain high performance on massively parallel computers. Such massively parallel machines also need massively parallel I/O.

There are difficulties in getting high performance I/O:

- Efficient programs must avoid sequential bottlenecks from processors to file systems
- Fortran specifies that a file appears in element storage order; this conflicts with striped files (for example, an array distributed by rows may be written to a file striped by columns).

In particular Fortran file organization has limits:

- Files have a sequential organization. (Even direct access files have records in sequential order, though they can be accessed out of order)
- Fortran files are record oriented
- Storage and sequence association are in force (when writing and then reading a file, for instance)
- No specification of the physical organization is possible
- No compatibility with other languages/machines is guaranteed

With these in mind there are two major approaches that have been suggested:

1. Define hints (annotations) that do not change file semantics, in the spirit of data distribution. (This gives some information to the compiler.)
2. Introduce parallel read/write operations that are not necessarily compatible with sequential ones.

13.1 Hints

Two ideas have been advanced which use the idea of giving hints to the compiler without changing the Fortran file semantics.

The first is based on the observation that although the distribution of an array when it is written may be available to the compiler or runtime system, the distribution into which

that array will be read cannot generally be known, even though the programmer may have this knowledge. So the proposal is to provide on a write a hint about how the data will be read.

```
!HPF$ DISTRIBUTE (CYCLIC) :: a
!HPF$ IO_DISTRIBUTE * :: a
WRITE a, b, c
!HPF$ IO_DISTRIBUTE * :: b
```

When an array is written, it can be easily read back in the given distribution. The annotation can be associated with either the declaration or the write itself; in the first case it applies to all writes of the array, while in the second it only applies to the one statement. The intent is that meta-data is kept in the file system to record the "right" data layout. The advantages of this proposal include notation and efficiency.

The second proposal is to give hints about the physical layout (number of spins, record length, striping function, etc.) of the file when it is opened.

This uses the HPF array mapping mechanisms. (A file is a 1-dimensional array of records.) The syntax needs a "name" for the file "template": we suggest FILEMAP. The programmer can align/distribute FILEMAP (on I/O nodes), associate FILEMAP with a file on OPEN, etc. There are no changes in semantics or file system.

Mapping Files A Fortran file is a sequence of records. We treat such file as a 1-D array of records with LB=1 and infinite UB. This array can be mapped to a (storage) node arrangement in a manner analogous to the mapping of an array to a (processor) node arrangement. Files are mapped using the same notation as for array mapping. The mapping defines a partition of the file, and each part is associated with one abstract node.

The mapping of a file to a node arrangement can be interpreted in two ways:

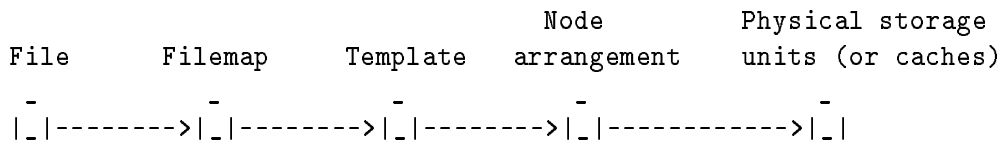
1. The nodes may represent (abstract) independent storage units, each storing a fixed part of the file.

2. The nodes may represent (abstract) independent file caches, with a fixed association of each cache with a part of the file.

In both cases the file is mapped onto physical I/O devices so as to allow maximal concurrency for accesses directed to distinct parts of the file. If the second interpretation is used, then it is meaningful to align arrays and files onto the same templates.

We introduce a new filemap object. Filemaps are, essentially, named files. They appear where an array name would appear in a array mapping expression. An actual file is associated with a FILEMAP in an OPEN statement. Filemaps are introduced because files are not first class objects in FORTRAN (files are not declared). Also, Filemaps can have rank > 1, giving more flexibility in the types of mappings that can be specified.

The following diagram illustrates the mapping



Node Directive We suggest to replace the keyword `PROCESSOR` with the keyword `NODE`, which is more neutral. Node arrangements (ex processor arrangements) can be targets both for file mappings and for array mappings. Some implementations may disallow the use of the same node arrangement name as a target both for array mappings and for file mappings. In such case an `AFFINITY` directive, that specifies affinity between io nodes and processor nodes, would be useful. (Such directive would also be useful to specify affinity between nodes of different arrangements, e.g. nodes in arrangements of different rank.)

The set of allowable node arrangements that can be used to map files is implementation dependent – however, a node arrangement with `NUMBER_OF_IONODES` nodes is always legal.

The mapping of nodes to physical storage units is implementation dependent.

For example:

```
!HPF$ NODE :: D1(2,4), D2(2,2)
      PARAMETER(NOD=NUMBER_OF_IONODES())
!HPF$ NODE, DIMENSION(NOD) :: D3,D4
```

FILEMAP Directive A Fortran file is an infinite one-dimensional array of records, with LB=1. A filemap can be thought of as an assumed-size array of records. This array is associated with (one-dimensional) files, using storage association rules. The filemap name is used to specify a mappings for files. The association between a filemap name and an actual file is effected by the OPEN statement.

A FILEMAP directive declares filemap names. The syntax is

J1301 *filemap-directive* **is** `FILEMAP [::]`
filemap-name `(` *assumed-size-spec* `)`
`[,` *filemap-name* `(`*assumed-size-spec* `) ] ...`
or `FILEMAP, DIMENSION (` *assumed-size-spec* `)`
`::` *filemap-name-list*

An *assumed-size-spec* is a specification of the form used for assumed sized arrays: All dimensions are specified, with the exception of the last, which is assumed. In our case, the last dimension is infinite. Only initialization expressions may occur in this specification (including expressions that depend on `NUMBER_OF_IONODES`).

For example:

```
!HPF$ FILEMAP :: F1(2,4,*)
!HPF$ FILEMAP, DIMENSION(2,2,1:*) :: F2,F3
```

A FILEMAP directive does not allocate space, neither in memory, nor on disk.

File mapping ALIGN and DISTRIBUTE statements are used to map FILEMAPs onto nodes. The syntax is identical to the syntax for processor mappings, with one restriction: Block distributions cannot be used for the last (infinite) dimension of the filemap.

For example:

```
!HPF$ DISTRIBUTE (CYCLIC,CYCLIC,*) ONTO D2 :: F2,F3
!HPF$ DISTRIBUTE F1(*,BLOCK,CYCLIC(2)) ONTO D1
```

Assume that `F1, F2` are the filemaps and `D1, D2` are the node arrangements from the previous examples.

The first distribute statement specifies the following mapping for successive records of a file associated with `F2` or `F3`.

```
D2(1,1)          D2(1,2)

1 (1,1,1)        3 (1,2,1)
5 (1,1,2)        7 (1,2,2)
.                .
.                .
.                .



D2(2,1)           D2(2,2)

2 (2,1,1)         4 (2,2,1)
6 (2,1,2)         8 (2,2,2)
.                 .
.                 .
```

The second distribute statement specifies the following mapping for successive records of a file associated with `F1`.

```
D1(1,1)           D1(1,2)          D1(1,3)           D1(1,4)

 1 (1,1,1)        17               33                49
 2 (2,1,1)        .                .                 .
 3 (1,2,1)        .                .                 .
 4 (2,2,1)        20               36                52
 9 (1,1,2)        25               41                57
10 (2,1,2)        .                .                 .
11 (1,2,2)                         .                 .
12 (2,2,2)        28               44                60
65                81               97               113
.                 .                .                 .
.                 .                .                 .
```

```
D1(2,1)          D1(2,2)        D1(2,3)          D1(2,4)

 5 (1,3,1)       21             37               53
 6 (2,3,1)       .              .                .
 7 (1,4,1)       .              .                .
 8 (2,4,1)       24             40               56
13 (1,3,2)       29             45               61
14 (2,3,2)       .              .                .
15 (1,4,2)       .              .                .
16 (2,4,2)       32             48               64
69               85            101              117
.                .              .                .
.                .              .                .
```

13.2 Explicit Parallel I/O Statements

13.2.1 OPEN statement

A new connection specifier of the form `FILEMAP = filemap-name` associates a mapping with the opened file. If the file exists then the mapping must be one of the mappings allowed for the file. The set of allowed file mappings for an existing file is implementation dependent, but always includes the mapping under which the file was created. More generally, it will include any mapping where the file is mapped onto the same storage node arrangement, and with the same allocations of file records to storage nodes (different mappings may result in the same allocation of records to storage nodes). One choice is to allow any mapping, with possible degraded performance for ill matched mappings; another choice is to remap an existing file when it is opened with a new mapping, either offline or online. Vendors are expected to provide implementation dependent mechanisms to exercise such choices.

The default mapping is implementation dependent.

Only external files can be mapped.

Implementations may restrict the use of the FILEMAP connection specifier to files that are open for direct access (i.e., fixed size record files).

13.2.2 Parallel Data Transfer

The READ, WRITE, CLOSE, INQUIRE, BACKSPACE, ENDFILE, REWIND statements can be used to access distributed files; there are no changes in the syntax or semantics of these statements.

PREAD and PWRITE statements are added to allow efficient input or output of distributed arrays. The PREAD and PWRITE statements have the same syntax as unformatted I/O statements with READ or WRITE, respectively; they are semantically different. The data representation created on a file by a PWRITE statement may be different from the data representation that obtains if PWRITE is replaced by WRITE. In particular, whereas an unformatted WRITE statement will create a single record (stored on one I/O

node), a PWRITE statement may create multiple records, possibly on multiple I/O nodes. Whereas an unformatted READ statement accesses a unique record, a PREAD statement may access multiple records.

If a PWRITE statement was used to write a list of output items on a file, then a PREAD that starts at the same point in the file, and has a compatible list of input items, will return the values that were written. Two lists of items are compatible if the corresponding items in each list occupy the same number of storage units and have compatible mappings (informally, if the distribution of entries onto abstract processors is the same).

The following program exchanges the values of arrays `A` and `B`. The exchange is legal because the arrays are compatible.

```
REAL, DIMENSION(1000,1000) :: A, B
!HPF$ ALIGN WITH B :: A
...
OPEN(UNIT = 15, ACTION = READWRITE)
PWRITE (UNIT = 15) A, B
REWIND (UNIT = 15)
PREAD (UNIT = 15) B, A
```

The behavior of the following program is undefined. More than one record could have been created by the PWRITE statement, so that the BACKSPACE statement does not necessarily return the file position to where it was before PWRITE executed.

```
REAL, DIMENSION(1000,1000) :: A, B
!HPF$ ALIGN WITH B :: A
OPEN(UNIT = 15, ACTION = READWRITE)
PWRITE (UNIT = 15) A, B
BACKSPACE (UNIT = 15)
PREAD (UNIT = 15) B, A
```

The behavior of the following program is undefined, since the two arrays `A` and `B` don't have compatible distributions.

```
REAL, DIMENSION(1000,1000) :: A, B
!HPF$ DISTRIBUTE A(BLOCK,BLOCK)
!HPF$ DISTRIBUTE B(CYCLIC, CYCLIC)
...
OPEN(UNIT = 15, ACTION = READWRITE)
PWRITE (UNIT = 15) A, B
REWIND (UNIT = 15)
PREAD (UNIT = 15) B, A
```

Data written by a WRITE statement cannot be read with PREAD, and data written with PWRITE cannot be read with READ, or by a PREAD that does not start at exactly the same point in the file (otherwise the program outcome is undefined).

PREAD and PWRITE can be used both for sequential access and for direct access. In the latter case, the REC specifier indicates the position in the file where from the transfer starts. It is still the case that a transfer may involve several records.

13.2.3 Restrictions

The following restrictions allow for a simpler, more efficient implementation of parallel I/O. We may either put them in the language, or list them as recommended programming style.

1. Items in the item list of a PREAD or PWRITE statements are restricted to be variables (no io-implied-do). [Compilers may want to relax this rule, by considering an io-implied-do as being an operation that defines a new variable, akin to an array section, with a distribution induced by the distribution of the variables appearing in the implied-do-loop.]

2. All values needed to determine which entities are specified by a parallel I/O item list need be specified before the I/O statement. That is, we prohibit a statement of the form `PREAD (...)  N, A(1:N)`.

13.2.4 Extensions

- We may want to write an array with a layout that is suited to the mapping of the array that will appear in the input item list, rather than suited to the mapping of the array in the output list. To achieve this, we need to add align/distribute information as part of the PWRITE statement.

- We may want a `REMAP` statement, to be used instead of the sequence `CLOSE ... OPEN`, in order to associate a new mapping to an existing file.

- We may want to extend the `INQUIRE` statement to return file mapping information. Alternatively, we may use the same query intrinsics used to query array partitions.

- A new intrinsic function of the form `INDEX(filemap-name, list-of-indices)` would be handy, in order to address random-access files as multi-dimensional arrays. E.g.

```
READ (7, REC = INDEX(F1,3,5) ) A
```

- Each data transfer operation specifies an association between parts of the file and abstract processor nodes where from (where to) the data in the record is transferred. We may want to add additional directives to the OPEN statement to indicate that this association fulfills certain restrictions for as long as the file is open.

  - Accesses to a file are *independent* if, in all data transfers, each file part is associated with the same processor node. An `INDEPENDENT` argument in the OPEN statement may be used to specify this condition (which simplifies file caching).
  - A data transfer is *aligned* if each file part is associated with a unique processor node (is not split among two processor nodes). We may use an `ALIGNED` argument in the OPEN statement to specify that all data transfers are aligned. (INDEPENDENT implies ALIGNED, but not vice versa).

Section 14

FORALL-ELSEFORALL construct

FORALL-ELSEFORALL construct is a natural generalization of Fortran 90 WHERE-ELSEWHERE construct. A construct proposed in previous drafts of the HPF:

```
FORALL(I=1:N,J=1:N)
  WHERE(MASK)
    assignment
  ELSEWHERE
    assignment
  ENDWHERE
ENDFORALL
```

seems to introduce unnecessary limitations coming from limitations of WHERE construct: the mask array must conform with the variables on the right side in all of the array assignment statements in the construct.

14.1 FORALL-ELSEFORALL Construct

The FORALL-ELSEFORALL construct is a generalization of the masked element array assignment statement allowing multiple assignments, masked array assignments, and nested FORALL statements to be controlled by a single *forall-triplet-spec-list*. Rule R215 for *executable-construct* is extended to include the *forall-construct*.

14.1.1 General Form of the FORALL-ELSEFORALL Construct

J1401 *forall-construct* **is** `FORALL` (*forall-triplet-spec-list*
[, *scalar-mask-expr*])
forall-body-stmt-list
[`ELSEFORALL`]
[*elseforall-body-stmt-list*]
`END FORALL`

J1402 *forall-body-stmt* **is** *forall-assignment*
or *forall-stmt*
or *forall-construct*

J1403 *elseforall-body-stmt* **is** *forall-body-stmt*

Constraint: *index-name* must be a *scalar-name* of type integer.

Constraint: A *subscript* or a *stride* in a *forall-triplet-spec* must not contain a reference to any *index-name* in the *forall-triplet-spec-list.*

Constraint: Any left-hand side *array-section* or *array-element* in any *forall-body-stmt* must reference all of the *forall-triplet-spec index-name*s.

Constraint: If a *forall-stmt* or *forall-construct* is nested within a *forall-construct*, then the inner FORALL may not redefine any *index-name* used in the outer *forall-construct.* This rule applies recursively in the event of multiple nesting levels.

For each index name in the *forall-assignment*s, the set of permitted values is determined on entry to the construct and is

$$m1 + (k-1) * m3, where\ k = 1, 2, ..., \lfloor \frac{(m2 - m1 + 1)}{m3} \rfloor$$

and where *m1*, *m2*, and *m3* are the values of the first subscript, the second subscript, and the stride respectively in the *forall-triplet-spec.* If *stride* is missing, it is as if it were present with a value of the integer 1. The expression *stride* must not have the value 0. If for some index name $\lfloor (m2 - m1 + 1)/m3 \rfloor \leq 0$, the *forall-assignment*s are not executed.

14.1.2 Interpretation of the FORALL Construct

Execution of a FORALL construct consists of the following steps:

1. Evaluation in any order of the subscript and stride expressions in the *forall-triplet-spec-list.* The set of valid combinations of *index-name* values is then the cartesian product of the sets defined by these triplets.

2. Evaluation of the *scalar-mask-expr* for all valid combinations of *index-name* values. The mask elements may be evaluated in any order. One set of active combinations of *index-name* values is the subset of the valid combinations for which the mask evaluates to true and a second one is the subset of the valid combinations for which the mask evaluates to false.

3. Execute *forall-body-stmts* in the order they appear for the set of the valid combination of *index-name* for which mask was evaluated to true in the step 2. Each statement is executed completely (that is, for all active combinations of *index-name* values) according to the following interpretation:

   (a) Assignment statements, pointer assignment statements, and array assignment statements (i.e. statements in the *forall-assignment* category) evaluate the right-hand side *expr* and any left-and side subscripts for all active *index-name* values, then assign those results to the corresponding left-hand side references.

   (b) FORALL statements and FORALL constructs first evaluate the subscript and stride expressions in the *forall-triplet-spec-list* for all active combinations of the outer FORALL constructs. The set of valid combinations of *index-name*s for

the inner FORALL is then the union of the sets defined by these bounds and strides for each active combination of the outer *index-name*s. The scalar mask expression is then evaluated for all valid combinations of the inner FORALL's *index-name*s to produce the set(s) of active combinations, as in step 2. If there is no scalar mask expression, it is assumed to be always true. Each statement in the inner FORALL is then executed for each valid combination (of the inner FORALL), recursively following the interpretations given in this section.

4. Execute *elseforall-body-stmts* for the set of active *index-name* for which the mask was evaluated to false in the step 2, the same way as in 3.

If the scalar mask expression is omitted, it is as if it were present with the value true. In that case ELSEFORALL statement is not allowed.

A single assignment or array assignment statement in a *forall-construct* must obey the same restrictions as a *forall-assignment* in a simple *forall-stmt*. (Note that the lowest level of nested statements must always be an assignment statement.) For example, an assignment may not cause the same array element to be assigned more than once. Different statements may, however, assign to the same array element, and assignments made in one statement may affect the execution of a later statement.

14.1.3 Scalarization of the FORALL-ELSEFORALL Construct

A *forall-construct* of the form:

```
FORALL ( v=l:u:s, mask )
    a(l:u:s) = rhs1
ELSEFORALL
    a(l:u:s) = rhs2
END FORALL
```

is equivalent to the following standard Fortran 90 code:

```
!evaluate subscript and stride expressions in any order

templ = l
tempu = u
temps = s

!then evaluate the masks

DO v1=templ,tempu,temps
    tempmask(v) = mask(v)
END DO

!then evaluate the first block of statements

DO v=templ,tempu,temps
  IF (tempmask(v)) THEN
```

```
      temprhs1(v) = rhs1
  END IF
END DO
DO v1=templ,tempu,temps
  IF (tempmask(v)) THEN
   a(v)=temprhs1(v)
  END IF
END DO

!then evaluate the second block of statements

DO v=templ,tempu,temps
  IF (not.tempmask(v)) THEN
      temprhs2(v) = rhs2
  END IF
END DO
DO v1=templ,tempu,temps
  IF (.not.tempmask(v)) THEN
   a(v)=temprhs2(v)
  END IF
END DO
```