# Section 3

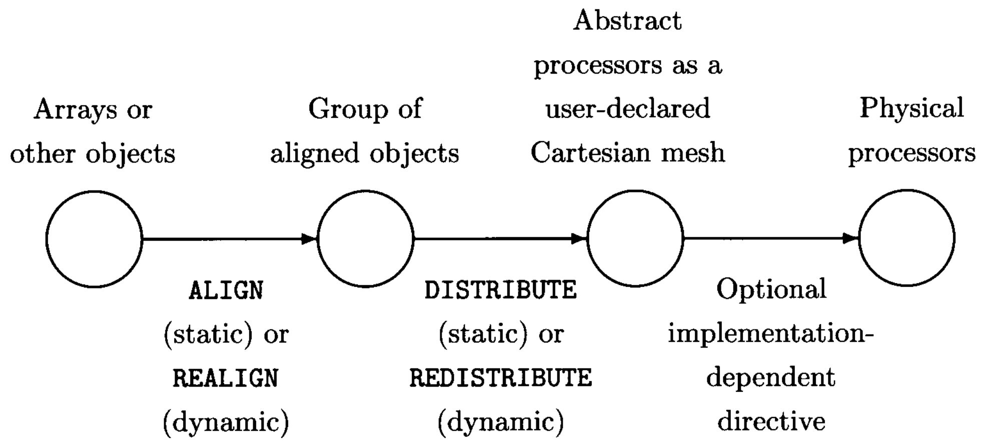
# Data Alignment and Distribution Directives

HPF data alignment and distributions directives allow the programmer to advise the compiler how to assign array elements to processor memories.

## 3.1 Model

HPF adds directives to Fortran 90 to allow the user to advise the compiler on the allocation of data objects to processor memories. The model is that there is a two-level mapping of data objects to memory regions, referred to as "abstract processors." Data objects (typically array elements) are first *aligned* relative to one another; this group of arrays is then *distributed* onto a rectilinear arrangement of abstract processors. (The implementation then uses the same number, or perhaps some smaller number, of physical processors to implement these abstract processors. This mapping of abstract processors to physical processors is language-processor dependent.)

The following diagram illustrates the model:


Arrays or other objects
Group of aligned objects
Abstract processors as a user-declared Cartesian mesh
Physical processors
ALIGN (static) or REALIGN (dynamic)
DISTRIBUTE (static) or REDISTRIBUTE (dynamic)
Optional implementation-dependent directive

The underlying assumptions are 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 may be possible to carry out many such operations concurrently if they can be performed on different processors.

Fortran 90 provides a number of features, notably array syntax, that make it easy for a compiler to determine that many operations may be carried out concurrently. The HPF directives provide a way to inform the compiler of the recommendation that certain data objects should reside in the same processor: if two data objects are mapped (via the two-level mapping of alignment and distribution) to the same abstract processor, it is a strong recommendation to the implementation that they ought to reside in the same physical processor. There is also a provision for recommending that a data object be stored in multiple locations, which may complicate any updating of the object but makes it faster for multiple processors to read the object.

There is a clear separation between directives that serve as specification statements and directives that serve as executable statements (in the sense of the Fortran standards). Specification statements are carried out on entry to a program unit, 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 contain specification expressions, which are permitted to depend on run-time quantities such as dummy arguments, and so the values of these expressions may not be available until run time, specifically the very moment that program control enters the scoping unit.)

The basic concept is that every array (indeed, every object) is created with *some* alignment to an entity, which in turn has *some* distribution onto *some* arrangement of abstract processors. If the specification statements contain explicit specification directives specifying the alignment of an array `A` with respect to another 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.

> *Advice to implementors.* This model gives a better picture of the actual amount of work that needs to be done than a model that says "the array is created in some default location, and then realigned and/or redistributed if there is an explicit directive." Using `ALIGN` and `DISTRIBUTE` specification directives doesn't have to cause any more work at run time than using the implementation defaults. (*End of advice to implementors.*)

In the case of an allocatable object, we say that the object is created whenever it is allocated. Specification directives for allocatable objects (and allocated pointer targets) may appear in the *specification-part* of a program unit, but take effect each time the array is created, rather than on entry to the scoping unit.

Alignment is considered an *attribute* (in the Fortran 90 sense) of a data object. If an object `A` is aligned (statically or dynamically) with an object `B`, which in turn is already aligned to an object `C`, this is regarded as an alignment of `A` with `C` directly, with `B` serving only as an intermediary at the time of specification. (This matters only in the case where `B` is subsequently realigned; the result is that `A` remains aligned with `C`.) We say that `A` is *immediately aligned* with `B` but *ultimately aligned* with `C`. If an object is not explicitly aligned with another object, we say that it is ultimately aligned with itself. The alignment relationships form a tree with everything ultimately aligned to the object at the root of the tree; however, the tree is always immediately "collapsed" so that every object is related directly to the root. Any object that is not a root can be explicitly realigned but not explicitly redistributed. Any object that is a root can be explicitly redistributed but must not be explicitly realigned if anything else is aligned to it.

Every object which is the root of an alignment tree has an associated *template* or index space. Typically, this template has the same rank and size in each dimension as the object associated with it. (The most important exception to this rule is dummy arguments with the `INHERIT` attribute, described in Section 3.9.) We often refer to "the template for an array," which means the template of the object to which the array is ultimately aligned. (When an explicit `TEMPLATE` (see Section 3.8) is used, this may be simply the template to which the array is explicitly aligned.)

The *distribution* step of the HPF model technically applies to the template of an array, although because of the close relationship noted above we often speak loosely of the distribution of an array. Distribution partitions the template among a set of abstract processors according to a given pattern. The combination of alignment (from arrays to templates) and distribution (from templates to processors) thus determines the relationship of an array to the processors; we refer to this relationship as the *mapping* of the array. (These remarks also apply to a scalar, which may be regarded as having an index space whose sole position is indicated by an empty list of subscripts.)

Every object is created as if according to some complete set of specification directives; if the program does not include complete specifications for the mapping of some object, the compiler provides defaults. By default an object is not aligned with any other object; it is ultimately aligned with itself. The default distribution is language-processor dependent, but must be expressible as explicit directives for that implementation. (The distribution of a sequential object must be expressible as explicit directives only if it is an aggregate cover (see Section 7).) Identically declared objects need not be provided with identical default distribution specifications; the compiler may, for example, take into account the contexts in which objects are used in executable code. The programmer may force identically declared objects to have identical distributions by specifying such distributions explicitly. (On the other hand, identically declared processor arrangements *are* guaranteed to represent "the same processors arranged the same way." This is discussed in more detail in Section 3.7.)

Once an object has been created, it can be remapped by realigning it or redistributing an object to which it is ultimately aligned; but communication may be required in moving the data around. Redistributing an object causes all objects then ultimately aligned with it also to be redistributed so as to maintain the alignment relationships.

Sometimes it is desirable 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 allows one to declare 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.

By analogy with the Fortran 90 `ALLOCATABLE` attribute, HPF includes the attribute `DYNAMIC`. It is not permitted to `REALIGN` an array that has not been declared `DYNAMIC`. Similarly, it is not permitted to `REDISTRIBUTE` an array or template that has not been declared `DYNAMIC`.

## 3.2 Syntax of Data Alignment and Distribution Directives

Specification directives in HPF have two forms: specification statements, analogous to the `DIMENSION` and `ALLOCATABLE` statements of Fortran 90; and an attribute form analogous to type declaration statements in Fortran 90 using the "`::`" punctuation.

The attribute form allows more than one attribute to be described in a single directive. HPF goes beyond Fortran 90 in not requiring that the first attribute, or indeed any of them, be a type specifier.

For syntactic convenience, the executable directives `REALIGN` and `REDISTRIBUTE` also come in two forms (statement form and attribute form) but may not be combined with other attributes in a single directive.

```
H301 combined-directive     is  combined-attribute-list :: entity-decl-list

H302 combined-attribute     is  ALIGN align-attribute-stuff
                            or  DISTRIBUTE dist-attribute-stuff
                            or  DYNAMIC
                            or  INHERIT
                            or  TEMPLATE
                            or  PROCESSORS
                            or  DIMENSION ( explicit-shape-spec-list )
```

Constraint: The same *combined-attribute* must not appear more than once in a given *combined-directive.*

Constraint: If the `DIMENSION` attribute appears in a *combined-directive*, any entity to which it applies must be declared with the HPF `TEMPLATE` or `PROCESSORS` type specifier.

The following rules constrain the declaration of various attributes, whether in separate directives or in a combined-directive.

The HPF keywords `PROCESSORS` and `TEMPLATE` play the role of type specifiers in declaring processor arrangements and templates. The HPF keywords `ALIGN`, `DISTRIBUTE`, `DYNAMIC`, and `INHERIT` play the role of attributes. Attributes referring to processor arrangements, to templates, or to entities with other types (such as `REAL`) may be combined in an HPF directive without having the type specifier appear.

Dimension information may be specified after an *object-name* or in a `DIMENSION` attribute. If both are present, the one after the *object-name* overrides the `DIMENSION` attribute (this is consistent with the Fortran 90 standard). For example, in:

```
!HPF$ TEMPLATE,DIMENSION(64,64) :: A,B,C(32,32),D
```

`A`, `B`, and `D` are $64 \times 64$ templates; `C` is $32 \times 32$.

A comment on asterisks: The asterisk character "*" appears in the syntax rules for HPF alignment and distribution directives in three distinct roles:

- When a lone asterisk appears as a member of a parenthesized list, it indicates either a collapsed mapping, wherein many elements of an array may be mapped to the same abstract processor, or a replicated mapping, wherein each element of an array may be mapped to many abstract processors. See the syntax rules for *align-source* and *align-subscript* (see Section 3.4) and for *dist-format* (see Section 3.3).

- When an asterisk appears before a left parenthesis "(" or after the keyword `WITH` or `ONTO`, it indicates that the directive constitutes an assertion about the *current* mapping of a dummy argument on entry to a subprogram, rather than a request for a *desired* mapping of that dummy argument. This use of the asterisk may appear *only* in directives that apply to dummy arguments (see Section 3.10).

- When an asterisk appears in an *align-subscript-use* expression, it represents the usual integer multiplication operator.

## 3.3 DISTRIBUTE and REDISTRIBUTE Directives

The DISTRIBUTE directive specifies a mapping of data objects to abstract processors in a processor arrangement. For example,

```
      REAL SALAMI(10000)
!HPF$ DISTRIBUTE SALAMI(BLOCK)
```

specifies that the array SALAMI should be distributed across some set of abstract processors by slicing it uniformly into blocks of contiguous elements. If there are 50 processors, the directive implies that the array should be divided into groups of 200 elements, with SALAMI(1:200) mapped to the first processor, SALAMI(201:400) mapped to the second processor, and so on. If there is only one processor, the entire array is mapped to that processor as a single block of 10000 elements.

The block size may be specified explicitly:

```
      REAL WEISSWURST(10000)
!HPF$ DISTRIBUTE WEISSWURST(BLOCK(256))
```

This specifies that groups of exactly 256 elements should be mapped to successive abstract processors. (There must be at least $\lceil 10000/256 \rceil = 40$ abstract processors if the directive is to be satisfied. The fortieth processor will contain a partial block of only 16 elements, namely WEISSWURST(9985:10000).)

HPF also provides a cyclic distribution format:

```
      REAL DECK_OF_CARDS(52)
!HPF$ DISTRIBUTE DECK_OF_CARDS(CYCLIC)
```

If there are 4 abstract processors, the first processor will contain DECK_OF_CARDS(1:49:4), the second processor will contain DECK_OF_CARDS(2:50:4), the third processor will contain DECK_OF_CARDS(3:51:4), and the fourth processor will contain DECK_OF_CARDS(4:52:4). Successive array elements are dealt out to successive abstract processors in round-robin fashion.

Distributions may be specified independently for each dimension of a multidimensional array:

```
      INTEGER CHESS_BOARD(8,8), GO_BOARD(19,19)
!HPF$ DISTRIBUTE CHESS_BOARD(BLOCK, BLOCK)
!HPF$ DISTRIBUTE GO_BOARD(CYCLIC,*)
```

The CHESS_BOARD array will be carved up into contiguous rectangular patches, which will be distributed onto a two-dimensional arrangement of abstract processors. The GO_BOARD array will have its rows distributed cyclically over a one-dimensional arrangement of abstract processors. (The "*" specifies that GO_BOARD is not to be distributed along its second axis; thus an entire row is to be distributed as one object. This is sometimes called "on-processor" distribution.)

The REDISTRIBUTE directive is similar to the DISTRIBUTE directive but is considered executable. An array (or template) may be redistributed at any time, provided it has been declared DYNAMIC (see Section 3.5). Any other arrays currently ultimately 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 (see Section 3.4). (This can require a lot of computational and communication effort at run time; the programmer must take care when using this feature.)

The `DISTRIBUTE` directive may appear only in the *specification-part* of a scoping unit. The `REDISTRIBUTE` directive may appear only in the *execution-part* of a scoping unit. The principal difference between `DISTRIBUTE` and `REDISTRIBUTE` is that `DISTRIBUTE` must contain only a *specification-expr* as the argument to a `BLOCK` or `CYCLIC` option, whereas in `REDISTRIBUTE` such an argument may be any integer expression. Another difference is that `DISTRIBUTE` is an attribute, and so can be combined with other attributes as part of a *combined-directive*, whereas `REDISTRIBUTE` is not an attribute (although a `REDISTRIBUTE` statement may be written in the style of attributed syntax, using "`::`" punctuation).

Formally, the syntax of the `DISTRIBUTE` and `REDISTRIBUTE` directives is:

| | | | |
|---|---|---|---|
| H303 | *distribute-directive* | **is** | `DISTRIBUTE` *distributee dist-directive-stuff* |
| H304 | *redistribute-directive* | **is** | `REDISTRIBUTE` *distributee dist-directive-stuff* |
| | | **or** | `REDISTRIBUTE` *dist-attribute-stuff* `::` *distributee-list* |
| H305 | *dist-directive-stuff* | **is** | *dist-format-clause* [ *dist-onto-clause* ] |
| H306 | *dist-attribute-stuff* | **is** | *dist-directive-stuff* |
| | | **or** | *dist-onto-clause* |
| H307 | *distributee* | **is** | *object-name* |
| | | **or** | *template-name* |
| H308 | *dist-format-clause* | **is** | ( *dist-format-list* ) |
| | | **or** | * ( *dist-format-list* ) |
| | | **or** | * |
| H309 | *dist-format* | **is** | `BLOCK` [ ( *int-expr* ) ] |
| | | **or** | `CYCLIC` [ ( *int-expr* ) ] |
| | | **or** | * |
| H310 | *dist-onto-clause* | **is** | `ONTO` *dist-target* |
| H311 | *dist-target* | **is** | *processors-name* |
| | | **or** | * *processors-name* |
| | | **or** | * |

Constraint: An *object-name* mentioned as a *distributee* must be a simple name and not a subobject designator.

Constraint: An *object-name* mentioned as a *distributee* may not appear as an *alignee* in an `ALIGN` or `REALIGN` directive.

Constraint: A *distributee* that appears in a `REDISTRIBUTE` directive must have the `DYNAMIC` attribute (see Section 3.5).

Constraint: If a *dist-format-list* is specified, its length must equal the rank of each *distributee.*

Constraint: If both a *dist-format-list* and a *processors-name* appear, the number of elements of the *dist-format-list* that are not "*" must equal the rank of the named processor arrangement.

Constraint: If a *processors-name* appears but not a *dist-format-list*, the rank of each *distributee* must equal the rank of the named processor arrangement.

Constraint: If either the *dist-format-clause* or the *dist-target* in a DISTRIBUTE directive begins with "*" then every *distributee* must be a dummy argument.

Constraint: Neither the *dist-format-clause* nor the *dist-target* in a REDISTRIBUTE may begin with "*".

Constraint: Any *int-expr* appearing in a *dist-format* of a DISTRIBUTE directive must be a *specification-expr*.

Note that the possibility of a DISTRIBUTE directive of the form

```
!HPF$ DISTRIBUTE dist-attribute-stuff :: distributee-list
```

is covered by syntax rule H301 for a *combined-directive*.

Examples:

```
!HPF$ DISTRIBUTE D1(BLOCK)
!HPF$ DISTRIBUTE (BLOCK,*,BLOCK) ONTO SQUARE:: D2,D3,D4
```

The meanings of the alternatives for *dist-format* are given below.

Define the ceiling division function CD(J,K) = (J+K-1)/K (using Fortran integer arithmetic with truncation toward zero.)

Define the ceiling remainder function CR(J,K) = J-K*CD(J,K).

The dimensions of a processor arrangement appearing as a *dist-target* are said to *correspond* in left-to-right order with those dimensions of a *distributee* for which the corresponding *dist-format* is not *. In the example above, processor arrangement SQUARE must be two-dimensional; its first dimension corresponds to the first dimensions of D2, D3, and D4 and its second dimension corresponds to the third dimensions of D2, D3, and D4.

Let $d$ be the size of a *distributee* in a certain dimension and let $p$ be the size of the processor arrangement in the corresponding dimension. For simplicity, assume all dimensions have a lower bound of 1. Then BLOCK($m$) means that a *distributee* position whose index along that dimension is $j$ is mapped to an abstract processor whose index along the corresponding dimension of the processor arrangement is CD($j$,$m$) (note that $m \times p \geq d$ must be true), and is position number $m$+CR($j$,$m$) among positions mapped to that abstract processor. The first *distributee* position in abstract processor $k$ along that axis is position number 1+$m$*($k$-1).

BLOCK by definition means the same as BLOCK(CD($d$,$p$)).

CYCLIC($m$) means that a *distributee* position whose index along that dimension is $j$ is mapped to an abstract processor whose index along the corresponding dimension of the processor arrangement is 1+MODULO(CD($j$,$m$)-1,$p$). The first *distributee* position in abstract processor $k$ along that axis is position number 1+$m$*($k$-1).

CYCLIC by definition means the same as CYCLIC(1).

CYCLIC($m$) and BLOCK($m$) imply the same distribution when $m \times p \geq d$, but BLOCK($m$) additionally asserts that the distribution will not wrap around in a cyclic manner, which a compiler cannot determine at compile time if $m$ is not constant. Note that CYCLIC and BLOCK (without argument expressions) do not imply the same distribution unless $p \geq d$, a degenerate case in which the block size is 1 and the distribution does not wrap around.

Suppose that we have 16 abstract processors and an array of length 100:

```
!HPF$ PROCESSORS SEDECIM(16)
      REAL CENTURY(100)
```

Distributing the array BLOCK (which in this case would mean the same as BLOCK(7)):

```
!HPF$ DISTRIBUTE CENTURY(BLOCK) ONTO SEDECIM
```

results in this mapping of array elements onto abstract processors:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 8 | 15 | 22 | 29 | 36 | 43 | 50 | 57 | 64 | 71 | 78 | 85 | 92 | 99 | |
| 2 | 9 | 16 | 23 | 30 | 37 | 44 | 51 | 58 | 65 | 72 | 79 | 86 | 93 | 100 | |
| 3 | 10 | 17 | 24 | 31 | 38 | 45 | 52 | 59 | 66 | 73 | 80 | 87 | 94 | | |
| 4 | 11 | 18 | 25 | 32 | 39 | 46 | 53 | 60 | 67 | 74 | 81 | 88 | 95 | | |
| 5 | 12 | 19 | 26 | 33 | 40 | 47 | 54 | 61 | 68 | 75 | 82 | 89 | 96 | | |
| 6 | 13 | 20 | 27 | 34 | 41 | 48 | 55 | 62 | 69 | 76 | 83 | 90 | 97 | | |
| 7 | 14 | 21 | 28 | 35 | 42 | 49 | 56 | 63 | 70 | 77 | 84 | 91 | 98 | | |

Distributing the array BLOCK(8):

```
!HPF$ DISTRIBUTE CENTURY(BLOCK(8)) ONTO SEDECIM
```

results in this mapping of array elements onto abstract processors:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 9 | 17 | 25 | 33 | 41 | 49 | 57 | 65 | 73 | 81 | 89 | 97 | | | |
| 2 | 10 | 18 | 26 | 34 | 42 | 50 | 58 | 66 | 74 | 82 | 90 | 98 | | | |
| 3 | 11 | 19 | 27 | 35 | 43 | 51 | 59 | 67 | 75 | 83 | 91 | 99 | | | |
| 4 | 12 | 20 | 28 | 36 | 44 | 52 | 60 | 68 | 76 | 84 | 92 | 100 | | | |
| 5 | 13 | 21 | 29 | 37 | 45 | 53 | 61 | 69 | 77 | 85 | 93 | | | | |
| 6 | 14 | 22 | 30 | 38 | 46 | 54 | 62 | 70 | 78 | 86 | 94 | | | | |
| 7 | 15 | 23 | 31 | 39 | 47 | 55 | 63 | 71 | 79 | 87 | 95 | | | | |
| 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 | 80 | 88 | 96 | | | | |

Distributing the array BLOCK(6) is not HPF-conforming because $6 \times 16 < 100$.

Distributing the array CYCLIC (which means exactly the same as CYCLIC(1)):

```
!HPF$ DISTRIBUTE CENTURY(CYCLIC) ONTO SEDECIM
```

results in this mapping of array elements onto abstract processors:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
| 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 |
| 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 |
| 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 | 96 |
| 97 | 98 | 99 | 100 | | | | | | | | | | | | |

Distributing the array `CYCLIC(3)`:

```
!HPF$ DISTRIBUTE CENTURY(CYCLIC(3)) ONTO SEDECIM
```

results in this mapping of array elements onto abstract processors:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 4 | 7 | 10 | 13 | 16 | 19 | 22 | 25 | 28 | 31 | 34 | 37 | 40 | 43 | 46 |
| 2 | 5 | 8 | 11 | 14 | 17 | 20 | 23 | 26 | 29 | 32 | 35 | 38 | 41 | 44 | 47 |
| 3 | 6 | 9 | 12 | 15 | 18 | 21 | 24 | 27 | 30 | 33 | 36 | 39 | 42 | 45 | 48 |
| 49 | 52 | 55 | 58 | 61 | 64 | 67 | 70 | 73 | 76 | 79 | 82 | 85 | 88 | 91 | 94 |
| 50 | 53 | 56 | 59 | 62 | 65 | 68 | 71 | 74 | 77 | 80 | 83 | 86 | 89 | 92 | 95 |
| 51 | 54 | 57 | 60 | 63 | 66 | 69 | 72 | 75 | 78 | 81 | 84 | 87 | 90 | 93 | 96 |
| 97 | 100 | | | | | | | | | | | | | | |
| 98 | | | | | | | | | | | | | | | |
| 99 | | | | | | | | | | | | | | | |

A `DISTRIBUTE` or `REDISTRIBUTE` directive must not cause any data object associated with the *distributee* via storage association (`COMMON` or `EQUIVALENCE`) to be mapped such that storage units of a scalar data object are split across more than one abstract processor. See Section 7 for further discussion of storage association.

The statement form of a `DISTRIBUTE` or `REDISTRIBUTE` directive may be considered an abbreviation for an attributed form that happens to mention only one *alignee*; for example,

!HPF$ DISTRIBUTE *distributee* ( *dist-format-list* ) ONTO *dist-target*

is equivalent to

!HPF$ DISTRIBUTE ( *dist-format-list* ) ONTO *dist-target* :: *distributee*

Note that, to prevent syntactic ambiguity, the *dist-format-clause* must be present in the statement form, so in general the statement form of the directive may not be used to specify the mapping of scalars.

If the *dist-format-clause* is omitted from the attributed form, then the language processor may make an arbitrary choice of distribution formats for each template or array. So the directive

```
!HPF$ DISTRIBUTE ONTO P :: D1,D2,D3
```

means the same as

```
!HPF$ DISTRIBUTE ONTO P :: D1
!HPF$ DISTRIBUTE ONTO P :: D2
!HPF$ DISTRIBUTE ONTO P :: D3
```

to which a compiler, perhaps taking into account patterns of use of D1, D2, and D3 within the code, might choose to supply three distinct distributions such as, for example,

```
!HPF$ DISTRIBUTE D1(BLOCK, BLOCK) ONTO P
!HPF$ DISTRIBUTE D2(CYCLIC, BLOCK) ONTO P
!HPF$ DISTRIBUTE D3(BLOCK(43),CYCLIC) ONTO P
```

Then again, the compiler might happen to choose the same distribution for all three arrays.

In either the statement form or the attributed form, if the ONTO clause is present, it specifies the processor arrangement that is the target of the distribution. If the ONTO clause is omitted, then a language-processor-dependent processor arrangement is chosen arbitrarily for each *distributee*. So, for example,

```
      REAL, DIMENSION(1000) :: ARTHUR, ARNOLD, LINUS, LUCY
!HPF$ PROCESSORS EXCALIBUR(32)
!HPF$ DISTRIBUTE (BLOCK) ONTO EXCALIBUR :: ARTHUR, ARNOLD
!HPF$ DISTRIBUTE (BLOCK) :: LINUS, LUCY
```

causes the arrays ARTHUR and ARNOLD to have the same mapping, so that corresponding elements reside in the same abstract processor, because they are the same size and distributed in the same way (BLOCK) onto the same processor arrangement (EXCALIBUR). However, LUCY and LINUS do not necessarily have the same mapping, because they might, depending on the implementation, be distributed onto differently chosen processor arrangements; so corresponding elements of LUCY and LINUS might not reside on the same abstract processor. (The ALIGN directive provides a way to ensure that two arrays have the same mapping without having to specify an explicit processor arrangement.)

## 3.4 ALIGN and REALIGN Directives

The ALIGN directive is used to specify that certain data objects are to be mapped 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 (because two objects that are aligned are intended to be mapped to the same abstract processor). The ALIGN directive is designed to make it particularly easy to specify explicit mappings for all the elements of an array at once. While objects can be aligned in some cases through careful use of matching DISTRIBUTE directives, ALIGN is more general and frequently more convenient.

The REALIGN directive is similar to the ALIGN directive but is considered executable. An array (or template) may be realigned at any time, provided it has been declared DYNAMIC (see Section 3.5) Unlike redistribution (see Section 3.3), realigning a data object does not cause any other object to be remapped. (However, realignment of even a single object, if it is large, could require a lot of computational and communication effort at run time; the programmer must take care when using this feature.)

The `ALIGN` directive may appear only in the *specification-part* of a scoping unit. The `REALIGN` directive is similar but may appear only in the *execution-part* of a scoping unit. The principal difference between `ALIGN` and `REALIGN` is that `ALIGN` must contain only a *specification-expr* as a *subscript* or in a *subscript-triplet*, whereas in `REALIGN` such subscripts may be any integer expressions. Another difference is that `ALIGN` is an attribute, and so can be combined with other attributes as part of a *combined-directive*, whereas `REALIGN` is not an attribute (although a `REALIGN` statement may be written in the style of attributed syntax, using "`::`" punctuation).

Formally, the syntax of `ALIGN` and `REALIGN` is as follows:

| | | | |
|---|---|---|---|
| H312 | *align-directive* | **is** | `ALIGN` *alignee align-directive-stuff* |
| H313 | *realign-directive* | **is** | `REALIGN` *alignee align-directive-stuff* |
| | | **or** | `REALIGN` *align-attribute-stuff* `::` *alignee-list* |
| H314 | *align-directive-stuff* | **is** | `(` *align-source-list* `)` *align-with-clause* |
| H315 | *align-attribute-stuff* | **is** | [ `(` *align-source-list* `)` ] *align-with-clause* |
| H316 | *alignee* | **is** | *object-name* |
| H317 | *align-source* | **is** | `:` |
| | | **or** | `*` |
| | | **or** | *align-dummy* |
| H318 | *align-dummy* | **is** | *scalar-int-variable* |

Constraint: An *object-name* mentioned as an *alignee* may not appear as a *distributee* in a `DISTRIBUTE` or `REDISTRIBUTE` directive.

Constraint: Any *alignee* that appears in a `REALIGN` directive must have the `DYNAMIC` attribute (see Section 3.5).

Constraint: The *align-source-list* (and its surrounding parentheses) must be omitted if the *alignee* is scalar. (In some cases this will preclude the use of the statement form of the directive.)

Constraint: If the *align-source-list* is present, its length must equal the rank of the alignee.

Constraint: An *align-dummy* must be a named variable.

Constraint: An object may not have both the `INHERIT` attribute and the `ALIGN` attribute. (However, an object with the `INHERIT` attribute may appear as an *alignee* in a `REALIGN` directive, provided that it does not appear as a *distributee* in a `DISTRIBUTE` or `REDISTRIBUTE` directive.)

Note that the possibility of an `ALIGN` directive of the form

`!HPF$ ALIGN` *align-attribute-stuff* `::` *alignee-list*

is covered by syntax rule H301 for a *combined-directive*.

The statement form of an `ALIGN` or `REALIGN` directive may be considered an abbreviation of an attributed form that happens to mention only one *alignee*:

`!HPF$ ALIGN` *alignee* `(` *align-source-list* `) WITH` *align-spec* is equivalent to

```
!HPF$ ALIGN ( align-source-list ) WITH align-spec :: alignee
```

If the *align-source-list* is omitted from the attributed form and the *alignees* are not scalar, the *align-source-list* is assumed to consist of a parenthesized list of ":" entries, equal in number to the rank of the *alignees*. Similarly, if the *align-subscript-list* is omitted from the *align-spec* in either form, it is assumed to consist of a parenthesized list of ":" entries, equal in number to the rank of the *align-target*. So the directive

```
!HPF$ ALIGN WITH B :: A1, A2, A3
```

means

```
!HPF$ ALIGN (:,:) WITH B(:,:) :: A1, A2, A3
```

which in turn means the same as

```
!HPF$ ALIGN A1(:,:) WITH B(:,:)
!HPF$ ALIGN A2(:,:) WITH B(:,:)
!HPF$ ALIGN A3(:,:) WITH B(:,:)
```

because an attributed-form directive that mentions more than one *alignee* is equivalent to a series of identical directives, one for each *alignee*; all *alignees* must have the same rank. With this understanding, we will assume below, for the sake of simplifying the description, that an ALIGN or REALIGN directive has a single *alignee*.

Each *align-source* corresponds to one axis of the *alignee*, and is specified as either ":" or "*" or a dummy variable:

- If it is ":", then positions along that axis will be spread out across the matching axis of the *align-spec* (see below).
- If it is "*", then that axis is *collapsed*: positions along that axis make no difference in determining the corresponding position within the *align-target*. (Replacing the "*" with a dummy variable name not used anywhere else in the directive would have the same effect; "*" is merely a convenience that saves the trouble of inventing a variable name and makes it clear that no dependence on that dimension is intended.)
- A dummy variable is considered to range over all valid index values for that dimension of the *alignee*.

The WITH clause of an ALIGN has the following syntax:

| | | | |
|---|---|---|---|
| H319 | *align-with-clause* | **is** | WITH *align-spec* |
| H320 | *align-spec* | **is** | *align-target* [ ( *align-subscript-list* ) ] |
| | | **or** | * *align-target* [ ( *align-subscript-list* ) ] |
| H321 | *align-target* | **is** | *object-name* |
| | | **or** | *template-name* |
| H322 | *align-subscript* | **is** | *int-expr* |
| | | **or** | *align-subscript-use* |
| | | **or** | *subscript-triplet* |
| | | **or** | * |

| | | | |
|---|---|---|---|
| H323 | *align-subscript-use* | **is** | [ [ *int-level-two-expr* ] *add-op* ] *align-add-operand* |
| | | **or** | *align-subscript-use add-op int-add-operand* |
| H324 | *align-add-operand* | **is** | [ *int-add-operand* * ] *align-primary* |
| | | **or** | *align-add-operand* * *int-mult-operand* |
| H325 | *align-primary* | **is** | *align-dummy* |
| | | **or** | ( *align-subscript-use* ) |
| H326 | *int-add-operand* | **is** | *add-operand* |
| H327 | *int-mult-operand* | **is** | *mult-operand* |
| H328 | *int-level-two-expr* | **is** | *level-2-expr* |

Constraint: If the *align-spec* in an ALIGN directive begins with "*" then every *alignee* must be a dummy argument.

Constraint: The *align-spec* in a REALIGN may not begin with "*".

Constraint: Each *align-dummy* may appear at most once in an *align-subscript-list.*

Constraint: An *align-subscript-use* expression may contain at most one occurrence of an *align-dummy.*

Constraint: An *align-dummy* may not appear anywhere in the *align-spec* except where explicitly permitted to appear by virtue of the grammar shown above. Paraphrased, one may construct an *align-subscript-use* by starting with an *align-dummy* and then doing additive and multiplicative things to it with any integer expressions that contain no *align-dummy.*

Constraint: A *subscript* in an *align-subscript* may not contain occurrences of any *align-dummy.*

Constraint: An *int-add-operand*, *int-mult-operand*, or *int-level-two-expr* must be of type integer.

The syntax rules for an *align-subscript-use* take account of operator precedence issues, but the basic idea is simple: an *align-subscript-use* is intended to be a linear function of a single occurrence of an *align-dummy.*

For example, the following *align-subscript-use* expressions are valid, assuming that J, K, and M are *align-dummys* and N is not an *align-dummy*:

```
J    J+1      3-K      2*M     N*M        100-3*M
-J   +J       -K+3     M+2**3  M+N        -(4*7+IOR(6,9))*K-(13-5/3)
M*2  N*(M-N)  2*(J+1)  5-K+3   10000-M*3  2*(3*(K-1)+13)-100
```

The following expressions are not valid *align-subscript-use* expressions:

```
J+J  J-J       3*K-2*K  M*(N-M)  2*J-3*J+J  2*(3*(K-1)+13)-K
J*J  J+K       3/K      2**M     M*K        K-3*M
K-J  IOR(J,1)  -K/3     M*(2+M)  M*(M-N)    2**(2*J-3*J+J)
```

The *align-spec* must contain exactly as many *subscript-triplets* as the number of colons (":") appearing in the *align-source-list*. These are matched up in corresponding left-to-right order, ignoring, for this purpose, any *align-source* that is not a colon and any *align-subscript* that is not a *subscript-triplet*. Consider a dimension of the *alignee* for which a colon appears as an *align-source* and let the lower and upper bounds of that array be $LA$ and $UA$. Let the corresponding subscript triplet be $LT$:$UT$:$ST$ or its equivalent. Then the colon could be replaced by a new, as-yet-unused dummy variable, say J, and the subscript triplet by the expression (J-$LA$)*$ST$+$LT$ without affecting the meaning of the directive. Moreover, the axes must conform, which means that

$$\max(0, UA - LA + 1) \;=\; \max(0, \lceil (UT - LT + 1)/ST \rceil)$$

must be true. (This is entirely analogous to the treatment of array assignment.)

To simplify the remainder of the discussion, we assume that every colon in the *align-source-list* has been replaced by new dummy variables in exactly the fashion just described, and that every "*" in the *align-source-list* has likewise been replaced by an otherwise unused dummy variable. For example,

```
!HPF$ ALIGN A(:,*,K,:,:,*) WITH B(31:,:,K+3,20:100:3)
```

may be transformed into its equivalent

```
!HPF$ ALIGN A(I,J,K,L,M,N) WITH B(I-LBOUND(A,1)+31,       &
!HPF$              L-LBOUND(A,4)+LBOUND(B,2),K+3,(M-LBOUND(A,5))*3+20)
```

with the attached requirements

```
SIZE(A,1) .EQ. UBOUND(B,1)-30
 SIZE(A,4) .EQ. SIZE(B,2)
SIZE(A,5) .EQ. (100-20+3)/3
```

Thus we need consider further only the case where every *align-source* is a dummy variable and no *align-subscript* is a *subscript-triplet*.

Each dummy variable is considered to range over all valid index values for the corresponding dimension of the *alignee*. Every combination of possible values for the index variables selects an element of the *alignee*. The *align-spec* indicates a corresponding element (or section) of the *align-target* with which that element of the *alignee* should be aligned; this indication may be a function of the index values, but the nature of this function is syntactically restricted (as discussed above) to linear functions in order to limit the complexity of the implementation. Each *align-dummy* variable may appear at most once in the *align-spec* and only in certain rigidly prescribed contexts. The result is that each *align-subscript* expression may contain at most one *align-dummy* variable and the expression is constrained to be a linear function of that variable. (Therefore skew alignments are not possible.)

An asterisk "*" as an *align-subscript* indicates a replicated representation. Each element of the *alignee* is aligned with every position along that axis of the *align-target*.

*Rationale.* It may seem strange to use "*" to mean both collapsing and replication; the rationale is that "*" always stands conceptually for a dummy variable that appears nowhere else in the statement and ranges over the set of indices for the indicated dimension. Thus, for example,

```
!HPF$ ALIGN A(:) WITH D(:,*)
```

means that a copy of A is aligned with every column of D, because it is conceptually equivalent to

*for every legitimate index j, align* A(:) *with* D(:,*j*)

just as

```
!HPF$ ALIGN A(:,*) WITH D(:)
```

is conceptually equivalent to

*for every legitimate index j, align* A(:,*j*) *with* D(:)

Note, however, that while HPF syntax allows

```
!HPF$ ALIGN A(:,*) WITH D(:)
```

to be written in the alternate form

```
!HPF$ ALIGN A(:,J) WITH D(:)
```

it does *not* allow

```
!HPF$ ALIGN A(:) WITH D(:,*)
```

to be written in the alternate form

```
!HPF$ ALIGN A(:) WITH D(:,J)
```

because that has another meaning (only a variable appearing in the *align-source-list* following the *alignee* is understood to be an *align-dummy*, so the current value of the variable J is used, thus aligning A with a single column of D).

Replication allows an optimizing compiler to arrange to read whichever copy is closest. (Of course, when a replicated data object is written, all copies must be updated, not just one copy. Replicated representations are very useful for use as small lookup tables, where it is much faster to have a copy in each physical processor but without giving it an extra dimension that is logically unnecessary to the algorithm.) (*End of rationale.*)

By applying the transformations given above, all cases of an *align-subscript* may be conceptually reduced to either an *int-expr* (not involving an *align-dummy*) or an *align-subscript-use* and the *align-source-list* may be reduced to a list of index variables with no "*" or ":". An *align-subscript-list* may then be evaluated for any specific combination of values for the *align-dummy* variables simply by evaluating each *align-subscript* as an expression. The resulting subscript values must be legitimate subscripts for the *align-target*. (This implies that the *alignee* is not allowed to "wrap around" or "extend past the edges" of an *align-target*.) The selected element of the *alignee* is then considered to be aligned with the indicated element of the *align-target*; more precisely, the selected element of the *alignee* is considered to be ultimately aligned with the same object with which the indicated element of the *align-target* is currently ultimately aligned (possibly itself).

Once a relationship of ultimate alignment is established, it persists, even if the ultimate *align-target* is redistributed, unless and until the *alignee* is realigned by a **REALIGN** directive, which is permissible only if the *alignee* has the **DYNAMIC** attribute.

More examples of **ALIGN** directives:

```
      INTEGER D1(N)
      LOGICAL D2(N,N)
      REAL, DIMENSION(N,N):: X,A,B,C,AR1,AR2A,P,Q,R,S
!HPF$ ALIGN X(:,*) WITH D1(:)
!HPF$ ALIGN (:,*) WITH D1:: A,B,C,AR1,AR2A
!HPF$ ALIGN WITH D2, DYNAMIC:: P,Q,R,S
```

Note that, in a *alignee-list*, the alignees must all have the same rank but need not all have the same shape; the extents need match only for dimensions that correspond to colons in the *align-source-list*. This turns out to be an extremely important convenience; one of the most common cases in current practice is aligning arrays that match in distributed ("parallel") dimensions but may differ in collapsed ("on-processor") dimensions:

```
      REAL A(3,N), B(4,N), C(43,N), Q(N)
!HPF$ DISTRIBUTE Q(BLOCK)
!HPF$ ALIGN (*,:) WITH Q:: A,B,C
```

Here there are processors (perhaps N of them) and arrays of different sizes (3, 4, 43) within each processor are required. As far as HPF is concerned, the numbers 3, 4, and 43 may be different, because those axes will be collapsed. Thus array elements with indices differing only along that axis will all be aligned with the same element of Q (and thus be specified as residing in the same processor).

In the following examples, each directive in the group means the same thing, assuming that corresponding axis upper and lower bounds match:

```
!Second axis of X is collapsed
!HPF$ ALIGN X(:,*) WITH D1(:)
!HPF$ ALIGN X(J,*) WITH D1(J)
!HPF$ ALIGN X(J,K) WITH D1(J)

!Replicated representation along second axis of D3
!HPF$ ALIGN X(:,:) WITH D3(:,*,:)
!HPF$ ALIGN X(J,K) WITH D3(J,*,K)

!Transposing two axes
!HPF$ ALIGN X(J,K) WITH D2(K,J)
!HPF$ ALIGN X(J,:) WITH D2(:,J)
!HPF$ ALIGN X(:,K) WITH D2(K,:)
!But there isn't any way to get rid of *both* index variables;
! the subscript-triplet syntax alone cannot express transposition.
```

```
!Reversing both axes
!HPF$ ALIGN X(J,K) WITH D2(M-J+1,N-K+1)
!HPF$ ALIGN X(:,:) WITH D2(M:1:-1,N:1:-1)

!Simple case
!HPF$ ALIGN X(J,K) WITH D2(J,K)
!HPF$ ALIGN X(:,:) WITH D2(:,:)
!HPF$ ALIGN (J,K) WITH D2(J,K):: X
!HPF$ ALIGN (:,:) WITH D2(:,:):: X
!HPF$ ALIGN WITH D2:: X
```

## 3.5 DYNAMIC Directive

The `DYNAMIC` attribute specifies that an object may be dynamically realigned or redistributed.

| | | | |
|---|---|---|---|
| H329 | *dynamic-directive* | **is** | `DYNAMIC` *alignee-or-distributee-list* |
| H330 | *alignee-or-distributee* | **is** | *alignee* |
| | | **or** | *distributee* |

Constraint: An object in `COMMON` may not be declared `DYNAMIC` and may not be aligned to an object (or template) that is `DYNAMIC`. (To get this kind of effect, Fortran 90 modules must be used instead of `COMMON` blocks.)

Constraint: An object with the `SAVE` attribute may not be declared `DYNAMIC` and may not be aligned to an object (or template) that is `DYNAMIC`.

A `REALIGN` directive may not be applied to an *alignee* that does not have the `DYNAMIC` attribute. A `REDISTRIBUTE` directive may not be applied to a *distributee* that does not have the `DYNAMIC` attribute.

A `DYNAMIC` directive may be combined with other directives, with the attributes stated in any order, consistent with the Fortran 90 attribute syntax.

Examples:

```
!HPF$ DYNAMIC A,B,C,D,E
!HPF$ DYNAMIC:: A,B,C,D,E
!HPF$ DYNAMIC, ALIGN WITH SNEEZY:: X,Y,Z
!HPF$ ALIGN WITH SNEEZY, DYNAMIC:: X,Y,Z
!HPF$ DYNAMIC, DISTRIBUTE(BLOCK, BLOCK) :: X,Y
!HPF$ DISTRIBUTE(BLOCK, BLOCK), DYNAMIC :: X,Y
```

The first two examples mean exactly the same thing. The next two examples mean exactly the same second thing. The last two examples mean exactly the same third thing.

The three directives

```
!HPF$ TEMPLATE A(64,64),B(64,64),C(64,64),D(64,64)
!HPF$ DISTRIBUTE(BLOCK, BLOCK) ONTO P:: A,B,C,D
!HPF$ DYNAMIC A,B,C,D
```

may be combined into a single directive as follows:

```
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK, BLOCK) ONTO P,   &
!HPF$   DIMENSION(64,64),DYNAMIC :: A,B,C,D
```

## 3.6 Allocatable Arrays and Pointers

A variable with the POINTER or ALLOCATABLE attribute may appear as an *alignee* in an ALIGN directive or as a *distributee* in a DISTRIBUTE directive. Such directives do not take effect immediately, however; they take effect each time the array is allocated by an ALLOCATE statement, rather than on entry to the scoping unit. The values of all specification expressions in such a directive are determined once on entry to the scoping unit and may be used multiple times (or not at all). For example:

```
      SUBROUTINE MILLARD_FILLMORE(N,M)
      REAL, ALLOCATABLE, DIMENSION(:) :: A, B
!HPF$ ALIGN B(I) WITH A(I+N)
!HPF$ DISTRIBUTE A(BLOCK(M*2))
      N = 43
      M = 91
      ALLOCATE(A(27))
      ALLOCATE(B(13))
      ...
```

The values of the expressions N and M*2 on entry to the subprogram are conceptually retained by the ALIGN and DISTRIBUTE directives for later use at allocation time. When the array A is allocated, it is distributed with a block size equal to the retained value of M*2, not the value 182. When the array B is allocated, it is aligned relative to A according to the retained value of N, not its new value 43.

Note that it would have been incorrect in the MILLARD_FILLMORE example to perform the two ALLOCATE statements in the opposite order. In general, when an object X is created it may be aligned to another object Y only if Y has already been created or allocated. The following example illustrates several related cases.

```
      SUBROUTINE WARREN_HARDING(P,Q)
      REAL P(:)
      REAL Q(:)
      REAL R(SIZE(Q))
      REAL, ALLOCATABLE :: S(:),T(:)
!HPF$ ALIGN P(I) WITH T(I)                         !Nonconforming
!HPF$ ALIGN Q(I) WITH *T(I)                        !Nonconforming
!HPF$ ALIGN R(I) WITH T(I)                         !Nonconforming
!HPF$ ALIGN S(I) WITH T(I)
      ALLOCATE(S(SIZE(Q)))                         !Nonconforming
      ALLOCATE(T(SIZE(Q)))
```

The ALIGN directives are not HPF-conforming because the array T has not yet been allocated at the time that the various alignments must take place. The four cases differ slightly in their details. The arrays P and Q already exist on entry to the subroutine, but because T is not yet allocated, one cannot correctly prescribe the alignment of P or describe the alignment of Q relative to T. (See Section 3.10 for a discussion of prescriptive and descriptive directives.) The array R is created on subroutine entry and its size can correctly depend on the SIZE of Q, but the alignment of R cannot be specified in terms of the alignment of T any more than its size can be specified in terms of the size of T. It *is* permitted to have an alignment directive for S in terms of T, because the alignment action does not take place until S is allocated; however, the first ALLOCATE statement is nonconforming because S needs to be aligned but at that point in time T is still unallocated.

If an ALLOCATE statement is immediately followed by REDISTRIBUTE and/or REALIGN directives, the meaning in principle is that the array is first created with the statically declared alignment, then immediately remapped. In practice there is an obvious optimization: create the array in the processors to which it is about to be remapped, in a single step. HPF implementors are strongly encouraged to implement this optimization and HPF programmers are encouraged to rely upon it. Here is an example:

```
      REAL,ALLOCATABLE(:,:) :: TINKER, EVERS
!HPF$ DYNAMIC :: TINKER, EVERS
      REAL, POINTER :: CHANCE(:)
!HPF$ DISTRIBUTE(BLOCK),DYNAMIC :: CHANCE
      ...
      READ 6,M,N
      ALLOCATE(TINKER(N*M,N*M))
!HPF$ REDISTRIBUTE TINKER(CYCLIC, BLOCK)
      ALLOCATE(EVERS(N,N))
!HPF$ REALIGN EVERS(:,:) WITH TINKER(M::M,1::M)
      ALLOCATE(CHANCE(10000))
!HPF$ REDISTRIBUTE CHANCE(CYCLIC)
```

While CHANCE is by default always allocated with a BLOCK distribution, it should be possible for a compiler to notice that it will immediately be remapped to a CYCLIC distribution. Similar remarks apply to TINKER and EVERS. (Note that EVERS is mapped in a thinly-spread-out manner onto TINKER; adjacent elements of EVERS are mapped to elements of TINKER separated by a stride M. This thinly-spread-out mapping is put in the lower left corner of TINKER, because EVERS(1,1) is mapped to TINKER(M,1).)

An array pointer may be used in REALIGN and REDISTRIBUTE as an *alignee*, *align-target*, or *distributee* if and only if it is currently associated with a whole array, not an array section. One may remap an object by using a pointer as an *alignee* or *distributee* only if the object was created by ALLOCATE but is not an ALLOCATABLE array.

Any directive that remaps an object constitutes an assertion on the part of the programmer that the remainder of program execution would be unaffected if all pointers associated with any portion of the object were instantly to acquire undefined pointer association status, except for the one pointer, if any, used to indicate the object in the remapping directive.

*Advice to implementors.* If HPF directives were ever to be absorbed as actual Fortran statements, the previous paragraph could be written as "Remapping an object causes all pointers associated with any portion of the object to have undefined pointer association status, except for the one pointer, if any, used to indicate the object in the remapping directive." The more complicated wording here is intended to avoid any implication that the remapping directives, in the form of structured comment annotations, have any effect on the execution semantics, as opposed to the execution speed, of the annotated program.) (*End of advice to implementors.*)

When an array is allocated, it will be aligned to an existing template if there is an explicit ALIGN directive for the allocatable variable. If there is no explicit ALIGN directive, then the array will be ultimately aligned with itself. It is forbidden for any other object to be ultimately aligned to an array at the time the array becomes undefined by reason of deallocation. All this applies regardless of whether the name originally used in the `ALLOCATE` statement when the array was created had the `ALLOCATABLE` attribute or the `POINTER` attribute.

## 3.7 PROCESSORS Directive

The `PROCESSORS` directive declares one or more rectilinear processor arrangements, specifying for each one its name, its rank (number of dimensions), and the extent in each dimension. It may appear only in the *specification-part* of a scoping unit. Every dimension of a processor arrangement must have nonzero extent; therefore a processor arrangement cannot be empty.

In the language of section 14.1.2 of the Fortran 90 standard, processor arrangements are local entities of class (1); therefore a processor arrangement may not have the same name as a variable, named constant, internal procedure, etc., in the same scoping unit. Names of processor arrangements obey the same rules for host and use association as other names in the long list in section 12.1.2.2.1 of the Fortran 90 standard.

If two processor arrangements have the same shape, then corresponding elements of the two arrangements are understood to refer to the same abstract processor. (It is anticipated that language-processor-dependent directives provided by some HPF implementations could overrule the default correspondence of processor arrangements that have the same shape.)

If directives collectively specify that two objects be mapped to the same abstract processor at a given instant during the program execution, the intent is that the two objects be mapped to the same physical processor at that instant.

The intrinsic functions `NUMBER_OF_PROCESSORS` and `PROCESSORS_SHAPE` may be used to inquire about the total number of actual physical processors used to execute the program. This information may then be used to calculate appropriate sizes for the declared abstract processor arrangements.

| | | | |
|---|---|---|---|
| H331 | *processors-directive* | is | `PROCESSORS` *processors-decl-list* |
| H332 | *processors-decl* | is | *processors-name* [ ( *explicit-shape-spec-list* ) ] |
| H333 | *processors-name* | is | *object-name* |

Examples:

```
!HPF$ PROCESSORS P(N)
!HPF$ PROCESSORS Q(NUMBER_OF_PROCESSORS()),       &
!HPF$            R(8,NUMBER_OF_PROCESSORS()/8)
!HPF$ PROCESSORS BIZARRO(1972:1997,-20:17)
!HPF$ PROCESSORS SCALARPROC
```

If no shape is specified, then the declared processor arrangement is conceptually scalar.

*Rationale.* A scalar processor arrangement may be useful as a way of indicating that certain scalar data should be kept together but need not interact strongly with distributed data. Depending on the implementation architecture, data distributed onto such a processor arrangement may reside in a single "control" or "host" processor (if the machine has one), or may reside in an arbitrarily chosen processor, or may be replicated over all processors. For target architectures that have a set of computational processors and a separate scalar host computer, a natural implementation is to map every scalar processor arrangement onto the host processor. For target architectures that have a set of computational processors but no separate scalar "host" computer, data mapped to a scalar processor arrangement might be mapped to some arbitrarily chosen computational processor or replicated onto all computational processors. (*End of rationale.*)

An HPF compiler is required to accept any PROCESSORS declaration in which the product of the extents of each declared processor arrangement is equal to the number of physical processors that would be returned by the call NUMBER_OF_PROCESSORS(). It must also accept all declarations of scalar PROCESSOR arrangements. Other cases may be handled as well, depending on the implementation.

For compatibility with the Fortran 90 attribute syntax, an optional "::" may be inserted. The shape may also be specified with the DIMENSION attribute:

```
!HPF$ PROCESSORS :: RUBIK(3,3,3)
!HPF$ PROCESSORS, DIMENSION(3,3,3) :: RUBIK
```

As in Fortran 90, an *explicit-shape-spec-list* in a *processors-decl* will override an explicit DIMENSION attribute:

```
!HPF$ PROCESSORS, DIMENSION(3,3,3) ::      &
!HPF$               RUBIK, RUBIKS_REVENGE(4,4,4), SOMA
```

Here RUBIKS_REVENGE is $4 \times 4 \times 4$ while RUBIK and SOMA are each $3 \times 3 \times 3$. (By the rules enunciated above, however, such a statement may not be completely portable because no HPF language processor is required to handle shapes of total sizes 27 and 64 simultaneously.)

Returning from a subprogram causes all processor arrangements declared local to that subprogram to become undefined. It is not HPF-conforming for any array or template to be distributed onto a processor arrangement at the time the processor arrangement becomes undefined unless at least one of two conditions holds:

- The array or template itself becomes undefined at the same time by virtue of returning from the subprogram.

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

*Rationale.* Note that second condition is slightly less stringent than requiring all expressions to be constant. This allows calls to NUMBER_OF_PROCESSORS or PROCESSORS_SHAPE to appear without violating the condition. (*End of rationale.*)

Variables in COMMON or having the SAVE attribute may be mapped to a locally declared processor arrangement, but because the first condition cannot hold for such variables (they don't become undefined), the second condition must be observed. This allows COMMON variables to work properly through the customary strategy of putting identical declarations in each scoping unit that needs to use them, while allowing the processor arrangements to which they may be mapped to depend on the value returned by NUMBER_OF_PROCESSORS.

*Advice to implementors.* It may be desirable to have a way for the user to specify at compile time the number of physical processors on which the program is to be executed. This might be specified either by a language-processor-dependent directive, for example, or through the programming environment (for example, as a UNIX command-line argument). Such facilities are beyond the scope of the HPF specification, but as food for thought we offer the following illustrative hypothetical examples:

```
!Declaration for multiprocessor by ABC Corporation
!ABC$ PHYSICAL PROCESSORS(8)
!Declaration for mpp by XYZ Incorporated
!XYZ$ PHYSICAL PROCESSORS(65536)
!Declaration for hypercube machine by PDQ Limited
!PDQ$ PHYSICAL PROCESSORS(2,2,2,2,2,2,2,2,2,2)
!Declaration for two-dimensional grid machine by TLA GmbH
!TLA$ PHYSICAL PROCESSORS(128,64)
!One of the preceding might affect the following
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS())
```

It may furthermore be desirable to have a way for the user to specify the precise mapping of the processor arrangement declared in a PROCESSORS statement to the physical processors of the executing hardware. Again, this might be specified either by a language-processor-dependent directive or through the programming environment (for example, as a UNIX command-line argument); such facilities are beyond the scope of the HPF specification, but as food for thought we offer the following illustrative hypothetical example:

```
!PDQ$ PHYSICAL PROCESSORS(2,2,2,2,2,2,2,2,2,2,2,2,2)
!HPF$ PROCESSORS G(8,64,16)
!PDQ$ MACHINE LAYOUT G(:GRAY(0:2),:GRAY(6:11),:BINARY(3:5,12))
```

This might specify that the first dimension of G should use hypercube axes 0, 1, 2 with a Gray-code ordering; the second dimension should use hypercube axes 6 through 11 with a Gray-code ordering; and the third dimension should use hypercube axes 3, 4, 5, and 12 with a binary ordering. (*End of advice to implementors.*)

## 3.8 TEMPLATE Directive

The TEMPLATE directive declares one or more templates, specifying for each the name, the rank (number of dimensions), and the extent in each dimension. It must appear in the *specification-part* of a scoping unit.

In the language of section 14.1.2 of the Fortran 90 standard, templates are local entities of class (1); therefore a template may not have the same name as a variable, named constant, internal procedure, etc., in the same scoping unit. Template names obey the rules for host and use association as other names in the list in section 12.1.2.2.1 of the Fortran 90 standard.

A template is simply an abstract space of indexed positions; it can be considered as an "array of nothings" (as compared to an "array of integers," say). A template may be used as an abstract *align-target* that may then be distributed.

| | | | |
|---|---|---|---|
| H334 | *template-directive* | **is** | TEMPLATE *template-decl-list* |
| H335 | *template-decl* | **is** | *template-name* [ ( *explicit-shape-spec-list* ) ] |
| H336 | *template-name* | **is** | *object-name* |

Examples:

```
!HPF$ TEMPLATE A(N)
!HPF$ TEMPLATE B(N,N), C(N,2*N)
!HPF$ TEMPLATE DOPEY(100,100),SNEEZY(24),GRUMPY(17,3,5)
```

If the "::" syntax is used, then the declared templates may optionally be distributed in the same *combined-directive*. In this case all templates declared by the directive must have the same rank so that the DISTRIBUTE attribute will be meaningful. The DIMENSION attribute may also be used.

```
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK,*) ::    &
!HPF$                                WHINEY(64,64),MOPEY(128,128)
!HPF$ TEMPLATE, DIMENSION(91,91) :: BORED,WHEEZY,PERKY
```

Templates are useful in the particular situation where one must align several arrays relative to one another but there is no need to declare a single array that spans the entire index space of interest. For example, one might want four $N \times N$ arrays aligned to the four corners of a template of size $(N+1) \times (N+1)$:

```
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK, BLOCK) :: EARTH(N+1,N+1)
      REAL, DIMENSION(N,N) :: NW, NE, SW, SE
!HPF$ ALIGN NW(I,J) WITH EARTH( I , J )
!HPF$ ALIGN NE(I,J) WITH EARTH( I ,J+1)
!HPF$ ALIGN SW(I,J) WITH EARTH(I+1, J )
!HPF$ ALIGN SE(I,J) WITH EARTH(I+1,J+1)
```

Templates may also be useful in making assertions about the mapping of dummy arguments (see Section 3.10).

Unlike arrays, templates cannot be in COMMON. So two templates declared in different scoping units will always be distinct, even if they are 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.

Templates are not passed through the 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, though it may be a copy (see Section 3.9). On exit from a subprogram, an HPF implementation arranges that the actual argument is aligned with the same template with which it was aligned before the call.

Returning from a subprogram causes all templates declared local to that subprogram to become undefined. It is not HPF-conforming for any variable to be aligned to a template at the time the template becomes undefined unless at least one of two conditions holds:

- The variable itself becomes undefined at the same time by virtue of returning from the subprogram.

- Whenever the subprogram is called, the template is always locally defined in the same way, with identical lower bounds, identical upper bounds, and identical distribution information (if any) onto identically defined processor arrangements (see Section 3.7).

  *Rationale.* (Note that this second condition is slightly less stringent than requiring all expressions to be constant. This allows calls to NUMBER_OF_PROCESSORS or PROCESSORS_SHAPE to appear without violating the condition.) (*End of rationale.*)

Variables in COMMON or having the SAVE attribute may be mapped to a locally declared template, but because the first condition cannot hold for such variable (they don't become undefined), the second condition must be observed.

## 3.9 INHERIT Directive

The INHERIT directive specifies that a dummy argument should be aligned to a copy of the template of the corresponding actual argument in the same way that the actual argument is aligned.

H337 *inherit-directive* **is** INHERIT *dummy-argument-name-list*

The INHERIT directive causes the named subprogram dummy arguments to have the INHERIT attribute. Only dummy arguments may have the INHERIT attribute. An object may not have both the INHERIT attribute and the ALIGN attribute. The INHERIT directive may only appear in a *specification-part* of a scoping unit.

The INHERIT attribute specifies that the template for a dummy argument should be inherited, by making a copy of the template of the actual argument. Moreover, the INHERIT attribute implies a default distribution of DISTRIBUTE * ONTO *. Note that this default distribution is not part of Subset HPF; if a program uses INHERIT, it must override the default distribution with an explicit mapping directive in order to conform to Subset HPF. See Section 3.10 for further exposition. If an explicit mapping directive appears for the dummy argument, thereby overriding the default distribution, then the actual argument must be a whole array or a regular array section; it may not be an expression of any other form

If none of the attributes INHERIT, ALIGN, and DISTRIBUTE is specified explicitly for a dummy argument, then the template of the dummy argument has the same shape as the dummy itself and the dummy argument is aligned to its template by the identity mapping.

An INHERIT directive may be combined with other directives, with the attributes stated in any order, more or less consistent with Fortran 90 attribute syntax.

Consider the following example:

```
      REAL DOUGH(100)
!HPF$ DISTRIBUTE DOUGH(BLOCK(10))
      CALL PROBATE( DOUGH(7:23:2) )
      ...
      SUBROUTINE PROBATE(BREAD)
      REAL BREAD(9)
```

```
!HPF$ INHERIT BREAD
```

The inherited template of BREAD has shape [100]; element BREAD(I) is aligned with element 5 + 2*I of the inherited template and, since BREAD does not appear in a prescriptive DISTRIBUTE directive, it has a BLOCK(10) distribution.

## 3.10 Alignment, Distribution, and Subprogram Interfaces

Mapping directives may be applied to dummy arguments in the same manner as for other variables; such directives may also appear in interface blocks. However, there are additional options that may be used only with dummy arguments: asterisks, indicating that a specification is descriptive rather than prescriptive, and the INHERIT attribute.

First, consider the rules for the caller. If there is an explicit interface for the called subprogram and that interface contains mapping directives (whether prescriptive or descriptive) for the dummy argument in question, the actual argument will be remapped if necessary to conform to the directives in the explicit interface. The template of the dummy will then be as declared in the interface. If there is no explicit interface, then actual arguments that are whole arrays or regular array sections may be remapped at the discretion of the language processor; the values of other expressions may be mapped in any manner at the discretion of the language processor.

*Rationale.* The caller is required to treat descriptive directives in an explicit interface as if they were prescriptive so that the directives in the interface may be an exact textual copy of the directives appearing in the subprogram. If the *caller* enforces descriptive directives as if they were prescriptive, then the descriptive directives in the *called* routine will in fact be correct descriptions. (*End of rationale.*)

In order to describe explicitly the distribution of a dummy argument, the template that is subject to distribution must be determined. A dummy argument always has a fresh template to which it is ultimately aligned; this template is constructed in one of three ways:

- If the dummy argument appears explicitly as an *alignee* in an ALIGN directive, its template is specified by the *align-target.*
- If the dummy argument is not explicitly aligned and does not have the INHERIT attribute, then the template has the same shape and bounds as the dummy argument; this is called the *natural template* for the dummy.
- If the dummy argument is not explicitly aligned and does have the INHERIT attribute, then the template is "inherited" from the actual argument according to the following rules:
  - If the actual argument is a whole array, the template of the dummy is a copy of the template with which the actual argument is ultimately aligned.
  - If the actual argument is a regular array section of array $A$, then the template of the dummy is a copy of the template with which $A$ is ultimately aligned.
  - If the actual argument is any other expression, the shape and distribution of the template may be chosen arbitrarily by the language processor (and therefore the programmer cannot know anything *a priori* about its distribution).

In all of these cases, we say that the dummy has an *inherited template* rather than a natural template.

Consider the following example:

```
      LOGICAL FRUG(128),TWIST(128)
!HPF$ PROCESSORS DANCE_FLOOR(16)
!HPF$ DISTRIBUTE (BLOCK) ONTO DANCE_FLOOR::FRUG,TWIST
      CALL TERPSICHORE(FRUG(1:40:3),TWIST(1:40:3))
```

The two array sections FRUG(1:40:3) and TWIST(1:40:3) are mapped onto abstract processors in the same manner:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | 25 | | | | | | | | | | | | |
| | 10 | | | 34 | | | | | | | | | | | |
| | | 19 | | | | | | | | | | | | | |
| 4 | | | 28 | | | | | | | | | | | | |
| | 13 | | | 37 | | | | | | | | | | | |
| | | 22 | | | | | | | | | | | | | |
| 7 | | | 31 | | | | | | | | | | | | |
| | 16 | | | 40 | | | | | | | | | | | |

However, the subroutine TERPSICHORE will view them in different ways because it inherits the template for the second dummy but not the first:

```
      SUBROUTINE TERPSICHORE(FOXTROT,TANGO)
      LOGICAL FOXTROT(:),TANGO(:)
!HPF$ INHERIT TANGO
```

Therefore the template of TANGO is a copy of the 128 element template of the whole array TWIST. The template is mapped like this:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 9 | 17 | 25 | 33 | 41 | 49 | 57 | 65 | 73 | 81 | 89 | 97 | 105 | 113 | 121 |
| 2 | 10 | 18 | 26 | 34 | 42 | 50 | 58 | 66 | 74 | 82 | 90 | 98 | 106 | 114 | 122 |
| 3 | 11 | 19 | 27 | 35 | 43 | 51 | 59 | 67 | 75 | 83 | 91 | 99 | 107 | 115 | 123 |
| 4 | 12 | 20 | 28 | 36 | 44 | 52 | 60 | 68 | 76 | 84 | 92 | 100 | 108 | 116 | 124 |
| 5 | 13 | 21 | 29 | 37 | 45 | 53 | 61 | 69 | 77 | 85 | 93 | 101 | 109 | 117 | 125 |
| 6 | 14 | 22 | 30 | 38 | 46 | 54 | 62 | 70 | 78 | 86 | 94 | 102 | 110 | 118 | 126 |
| 7 | 15 | 23 | 31 | 39 | 47 | 55 | 63 | 71 | 79 | 87 | 95 | 103 | 111 | 119 | 127 |
| 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 | 80 | 88 | 96 | 104 | 112 | 120 | 128 |

TANGO(I) is aligned with element 3*I-2 of the template. But the template of FOXTROT has the same size 14 as FOXTROT itself. The actual argument, FRUG(1:40:3) is mapped to the 16 processors in this manner:

| Abstract processor | Elements of FRUG |
|---|---|
| 1 | 1, 2, 3 |
| 2 | 4, 5, 6 |
| 3 | 7, 8 |
| 4 | 9, 10, 11 |
| 5 | 12, 13, 14 |
| 6–16 | none |

It would be reasonable to understand the mapping of the template of FOXTROT to coincide with the layout of the array section:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 |  |  | 9 |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 4 |  |  | 12 |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 |  |  | 10 |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 5 |  |  | 13 |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 8 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3 |  |  | 11 |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 6 |  |  | 14 |  |  |  |  |  |  |  |  |  |  |  |

but we shall see that this is not permitted in HPF. Within subroutine TERPSICHORE it would be correct to make the descriptive assertion

```
!HPF$ DISTRIBUTE TANGO *(BLOCK)
```

but it would not be correct to declare

```
!HPF$ DISTRIBUTE FOXTROT *(BLOCK)                    !Nonconforming
```

Each of these asserts that the template of the specified dummy argument is already distributed BLOCK on entry to the subroutine. The shape of the template for TANGO is [128], inherited (copied) from the array TWIST, whose section was passed as the corresponding actual argument, and that template does indeed have a BLOCK distribution. But the shape of the template for FOXTROT is [14]; the layout of the elements of the actual argument FRUG(1:40:3) (3 on the first processor, 3 on the second processor, 2 on the third processor, 3 on the fourth processor, ...) cannot properly be described as a BLOCK distribution of a length-14 template, so the DISTRIBUTE declaration for FOXTROT shown above would indeed be erroneous.

On the other hand, the layout of FRUG(1:40:3) can be described in terms of an alignment to a length-128 template which can be described by an explicit TEMPLATE declaration (see Section 3.8), so the directives

```
!HPF$ PROCESSORS DANCE_FLOOR(16)
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK) ONTO DANCE_FLOOR::GURF(128)
!HPF$ ALIGN FOXTROT(J) WITH *GURF(3*J-2)
```

could be correctly included in TERPSICHORE to describe the layout of FOXTROT on entry to the subroutine without using an inherited template.

The simplest case is the use of the INHERIT attribute alone. If a dummy argument has the INHERIT attribute and no explicit ALIGN or DISTRIBUTE attribute, the net effect is to tell the compiler to leave the data exactly where it is—and not attempt to remap the actual argument. The dummy argument will be mapped in exactly the same manner as the actual argument; the subprogram must be compiled in such a way as to work correctly no matter how the actual argument may be mapped onto abstract processors. (It has this effect because an INHERIT attribute on a dummy D implicitly specifies the default distribution

```
!HPF$ DISTRIBUTE D * ONTO *
```

rather than allowing the compiler to choose any distribution it pleases for the dummy argument. The meaning of this implied DISTRIBUTE directive is discussed below.)

In the general case of a DISTRIBUTE directive, where every *distributee* is a dummy argument, either the *dist-format-clause* or the *dist-target*, or both, may begin with, or consist of, an asterisk.

- Without an asterisk, a *dist-format-clause* or *dist-target* is prescriptive; the clause describes a distribution and constitutes a request of the language processor to make it so. This might entail remapping or copying the actual argument at run time in order to satisfy the requested distribution for the dummy.

- Starting with an asterisk, a *dist-format-clause* or *dist-target* is descriptive; the clause describes a distribution and constitutes an assertion to the language processor that it will already be so. The programmer claims that, for every call to the subprogram, the actual argument will be such that the stated distribution already describes the mapping of that data. (The intent is that if the argument is passed by reference, no movement of the data will be necessary at run time. All this is under the assumption that the language processor has observed all other directives. While a conforming HPF language processor is not required to obey mapping directives, it should handle descriptive directives with the understanding that their implied assertions are relative to this assumption.)

- Consisting of only an asterisk, a *dist-format-clause* or *dist-target* is transcriptive; the clause says nothing about the distribution but constitutes a request of the language processor to copy that aspect of the distribution from that of the actual argument. (The intent is that if the argument is passed by reference, no movement of the data will be necessary at run time.) Note that the transcriptive case, whether explicit or implicit, is not included in Subset HPF.

It is possible that, in a single DISTRIBUTE directive, the *dist-format-clause* might have an asterisk but not the *dist-target*, or vice versa.

These examples of DISTRIBUTE directives for dummy arguments illustrate the various combinations:

```
!HPF$ DISTRIBUTE URANIA (CYCLIC) ONTO GALILEO
```

The language processor should do whatever it takes to cause URANIA to have a CYCLIC distribution on the processor arrangement GALILEO.

```
!HPF$ DISTRIBUTE POLYHYMNIA * ONTO ELVIS
```

The language processor should do whatever it takes to cause `POLYHYMNIA` to be distributed onto the processor arrangement `ELVIS`, using whatever distribution format it currently has (which might be on some other processor arrangement). (You can't say this in Subset HPF.)

```
!HPF$ DISTRIBUTE THALIA *(CYCLIC) ONTO FLIP
```

The language processor should do whatever it takes to cause `THALIA` to have a `CYCLIC` distribution on the processor arrangement `FLIP`; `THALIA` already has a cyclic distribution, though it might be on some other processor arrangement.

```
!HPF$ DISTRIBUTE CALLIOPE (CYCLIC) ONTO *HOMER
```

The language processor should do whatever it takes to cause `CALLIOPE` to have a `CYCLIC` distribution on the processor arrangement `HOMER`; `CALLIOPE` is already distributed onto `HOMER`, though it might be with some other distribution format.

```
!HPF$ DISTRIBUTE MELPOMENE * ONTO *EURIPIDES
```

`MELPOMENE` is asserted to already be distributed onto `EURIPIDES`; use whatever distribution format the actual argument had so, if possible, no data movement should occur. (You can't say this in Subset HPF.)

```
!HPF$ DISTRIBUTE CLIO *(CYCLIC) ONTO *HERODOTUS
```

`CLIO` is asserted to already be distributed `CYCLIC` onto `HERODOTUS` so, if possible, no data movement should occur.

```
!HPF$ DISTRIBUTE EUTERPE (CYCLIC) ONTO *
```

The language processor should do whatever it takes to cause `EUTERPE` to have a `CYCLIC` distribution onto whatever processor arrangement the actual was distributed onto. (You can't say this in Subset HPF.)

```
!HPF$ DISTRIBUTE ERATO * ONTO *
```

The mapping of `ERATO` should not be changed from that of the actual argument. (You can't say this in Subset HPF.)

```
!HPF$ DISTRIBUTE ARTHUR_MURRAY *(CYCLIC) ONTO *
```

`ARTHUR_MURRAY` is asserted to already be distributed `CYCLIC` onto whatever processor arrangement the actual argument was distributed onto, and no data movement should occur. (You can't say this in Subset HPF.)

Please note that `DISTRIBUTE ERATO * ONTO *` does not mean the same thing as

```
!HPF$ DISTRIBUTE ERATO *(*) ONTO *
```

This latter means: `ERATO` is asserted to already be distributed * (that is, on-processor) onto whatever processor arrangement the actual was distributed onto. Note that the processor arrangement is necessarily scalar in this case.

One may omit either the *dist-format-clause* or the *dist-target-clause* for a dummy argument. If such a clause is omitted and the dummy argument has the `INHERIT` attribute, then the compiler must handle the directive as if * or `ONTO *` had been specified explicitly. If such a clause is omitted and the dummy does not have the `INHERIT` attribute, then the compiler may choose the distribution format or a target processor arrangement arbitrarily. Examples:

```
!HPF$ DISTRIBUTE WHEEL_OF_FORTUNE *(CYCLIC)
```

WHEEL_OF_FORTUNE is asserted to already be CYCLIC. As long as it is kept CYCLIC, it may be remapped it onto some other processor arrangement, but there is no reason to.

```
!HPF$ DISTRIBUTE ONTO *TV :: DAVID_LETTERMAN
```

DAVID_LETTERMAN is asserted to already be distributed on TV in some fashion. The distribution format may be changed as long as DAVID_LETTERMAN is kept on TV. (Note that this declaration must be made in attributed form; the statement form

```
!HPF$ DISTRIBUTE DAVID_LETTERMAN ONTO *TV          !Nonconforming
```

does not conform to the syntax for a DISTRIBUTE directive.)

The asterisk convention allows the programmer to make claims about the pre-existing distribution of a dummy based on knowledge of the mapping of the actual argument. But what claims may the programmer correctly make?

If the dummy argument has an inherited template, then the subprogram may contain directives corresponding to the directives describing the actual argument. Sometimes it is necessary, as an alternative, to introduce an explicit named template (using a TEMPLATE directive) rather than inheriting a template; an example of this (GURF) appears above, near the beginning of this section.

If the dummy argument has a natural template (no INHERIT attribute) then things are more complicated. In certain situations the programmer is justified in inferring a pre-existing distribution for the natural template from the distribution of the actual's template, that is, the template that would have been inherited if the INHERIT attribute had been specified. In all these situations, the actual argument must be a whole array or array section, and the template of the actual must be coextensive with the array along any axes having a distribution format other than "*."

If the actual argument is a whole array, then the pre-existing distribution of the natural template of the dummy is identical to that of the actual argument.

If the actual argument is an array section, then, from each *section-subscript* and the distribution format for the corresponding axis of the array being subscripted, one constructs an axis distribution format for the corresponding axis of the natural template:

- If the *section-subscript* is scalar and the array axis is collapsed (as by an ALIGN directive) then no entry should appear in the distribution for the natural template.
- If the *section-subscript* is a *subscript-triplet* and the array axis is collapsed (as by an ALIGN directive), then * should appear in the distribution for the natural template.
- If the *section-subscript* is scalar and the array axis corresponds to an actual template axis distributed *, then no entry should appear in the distribution for the natural template.
- If the *section-subscript* is a *subscript-triplet* and the array axis corresponds to an actual template axis distributed *, then * should appear in the distribution for the natural template.
- If the *section-subscript* is a *subscript-triplet* $l\!:\!u\!:\!s$ and the array axis corresponds to an actual template axis distributed BLOCK($n$) (which might have been specified as

simply BLOCK, but there will be some $n$ that describes the resulting distribution) and *LB* is the lower bound for that axis of the array, then BLOCK($n/s$) should appear in the distribution for the natural template, *provided* that $s$ divides $n$ evenly and that $l - LB < s$.

- If the *section-subscript* is a *subscript-triplet* $l{:}u{:}s$ and the array axis corresponds to an actual template axis distributed CYCLIC($n$) (which might have been specified as simply CYCLIC, in which case $n = 1$) and *LB* is the lower bound for that axis of the array, then CYCLIC($n/s$) should appear in the distribution for the natural template, *provided* that $s$ divides $n$ evenly and that $l - LB < s$.

If the situation of interest is not described by the cases listed above, no assertion about the distribution of the natural template of a dummy is HPF-conforming.

Here is a typical example of the use of this feature. The main program has a two-dimensional array TROGGS, which is to be processed by a subroutine one column at a time. (Perhaps processing the entire array at once would require prohibitive amounts of temporary space.) Each column is to be distributed across many processors.

```
      REAL TROGGS(1024,473)
!HPF$ DISTRIBUTE TROGGS(BLOCK,*)
      DO J=1,473
        CALL WILD_THING(TROGGS(:,J))
      END DO
```

Each column of TROGGS has a BLOCK distribution. The rules listed above justify the programmer in saying so:

```
      SUBROUTINE WILD_THING(GROOVY)
      REAL GROOVY(:)
!HPF$ DISTRIBUTE GROOVY *(BLOCK) ONTO *
```

Consider now the ALIGN directive. The presence or absence of an asterisk at the start of an *align-spec* has the same meaning as in a *dist-format-clause*: it specifies whether the ALIGN directive is descriptive or prescriptive, respectively.

If an *align-spec* that does not begin with * is applied to a dummy argument, the meaning is that the dummy argument will be forced to have the specified alignment on entry to the subprogram (which may require temporarily remapping the data of the actual argument or a copy thereof).

Note that a dummy argument may also be used as an *align-target.*

```
      SUBROUTINE NICHOLAS(TSAR,CZAR)
      REAL, DIMENSION(1918) :: TSAR,CZAR
!HPF$ INHERIT :: TSAR
!HPF$ ALIGN WITH TSAR :: CZAR
```

In this example the first dummy argument, TSAR, is allowed to remain aligned with the corresponding actual argument, while the second dummy argument, CZAR, is forced to be aligned with the first dummy argument. If the two actual arguments are already aligned, no remapping of the data will be required at run time; but the subprogram will operate correctly even if the actual arguments are not already aligned, at the cost of remapping the data for the second dummy argument at run time.

If the *align-spec* begins with "*", then the *alignee* must be a dummy argument and the directive must be ALIGN and not REALIGN. The "*" indicates that the ALIGN directive constitutes a guarantee on the part of the programmer that, on entry to the subprogram, the indicated alignment will already be satisfied by the dummy argument, without any action to remap it required at run time. For example:

```
      SUBROUTINE GRUNGE(PLUNGE,SPONGE)
      REAL PLUNGE(1000),SPONGE(1000)
!HPF$ ALIGN PLUNGE WITH *SPONGE
```

This asserts that, for every J in the range 1:1000, on entry to subroutine GRUNGE, the directives in the program have specified that PLUNGE(J) is currently mapped to the same abstract processor as SPONGE(J). (The intent is that if the language processor has in fact honored the directives, then no interprocessor communication will be required to achieve the specified alignment.)

The alignment of a general expression is up to the language processor and therefore unpredictable by the programmer; but the alignment of whole arrays and array sections is predictable. In the code fragment

```
      REAL FIJI(5000),SQUEEGEE(2000)
!HPF$ ALIGN SQUEEGEE(K) WITH FIJI(2*K)
      CALL GRUNGE(FIJI(2002:4000:2),SQUEEGEE(1001:))
```

it is true that every element of the array section SQUEEGEE(1001:) is aligned with the corresponding element of the array section FIJI(2002:4000:2), so the claim made in subroutine GRUNGE is satisfied by this particular call.

It is not permitted to say simply "ALIGN WITH *"; an *align-target* must follow the asterisk. (The proper way to say "accept any alignment" is INHERIT.)

If a dummy argument has no explicit ALIGN or DISTRIBUTE attribute, then the compiler provides an implicit alignment and distribution specification, one that could have been described explicitly without any "assertion asterisks".

The rules on the interaction of the REALIGN and REDISTRIBUTE directives with a subprogram argument interface are:

1. A dummy argument may be declared DYNAMIC. However, it is subject to the general restrictions concerning the use of the name of an array to stand for its associated template.

2. If an array or any section thereof is accessible by two or more paths, it is not HPF-conforming to remap it through any of those paths. For example, if an array is passed as an actual argument, it is forbidden to realign that array, or to redistribute an array or template to which it was aligned at the time of the call, until the subprogram has returned from the call. This prevents nasty aliasing problems. An example:

```
      MODULE FOO
      REAL A(10,10)
!HPF$ DYNAMIC ::  A
      END

      PROGRAM MAIN
```

```
            USE FOO
            CALL SUB(A(1:5,3:9))
            END

            SUBROUTINE SUB(B)
            USE FOO
            REAL B(:,:)
            ...
      !HPF$ REDISTRIBUTE A             !nonconforming
            ...
            END
```

Situations such as this are forbidden, for the same reasons that an assignment to A at the statement marked "nonconforming" would also be forbidden. In general, in *any* situation where assignment to a variable would be nonconforming by reason of aliasing, remapping of that variable by an explicit REALIGN or REDISTRIBUTE directive is also forbidden.

An overriding principle is that any mapping or remapping of arguments is not visible to the caller. This is true whether such remapping is implicit (in order to conform to prescriptive directives, which may themselves be explicit or implicit) or explicit (specified by REALIGN or REDISTRIBUTE directives). When the subprogram returns and the caller resumes execution, all objects accessible to the caller after the call are mapped exactly as they were before the call. It is not possible for a subprogram to change the mapping of any object in a manner visible to its caller, not even by means of REALIGN and REDISTRIBUTE.

*Advice to implementors.* There are several implementation strategies for achieving this behavior. For example, one may be able to use a copy-in/copy-out strategy for arguments that require remapping on subprogram entry. Alternatively, one may be able to remap the actual argument on entry and remap again on exit to restore the original mapping. (*End of advice to implementors.*)