

1
2
3
4
5
6 **Section 3**
7
8

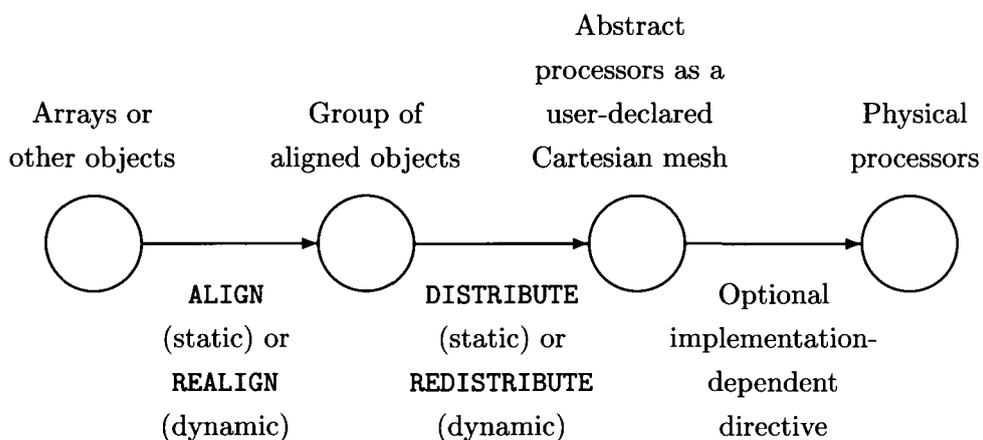
9 **Data Alignment and Distribution**
10 **Directives**
11
12
13
14
15

16 HPF data alignment and distributions directives allow the programmer to advise the com-
17 piler how to assign array elements to processor memories.
18

19 **3.1 Model**
20

21 HPF adds directives to Fortran 90 to allow the user to advise the compiler on the allocation
22 of data objects to processor memories. The model is that there is a two-level mapping
23 of data objects to memory regions, referred to as “abstract processors.” Data objects
24 (typically array elements) are first *aligned* relative to one another; this group of arrays is then
25 *distributed* onto a rectilinear arrangement of abstract processors. (The implementation then
26 uses the same number, or perhaps some smaller number, of physical processors to implement
27 these abstract processors. This mapping of abstract processors to physical processors is
28 language-processor dependent.)
29

30 The following diagram illustrates the model:



45 The underlying assumptions are that an operation on two or more data objects is
46 likely to be carried out much faster if they all reside in the same processor, and that it may
47 be possible to carry out many such operations concurrently if they can be performed on
48 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.

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

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

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

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

32 Sometimes it is desirable to consider a large index space with which several smaller
33 arrays are to be aligned, but not to declare any array that spans the entire index space.
34 HPF allows one to declare a **TEMPLATE**, which is like an array whose elements have no
35 content and therefore occupy no storage; it is merely an abstract index space that can be
36 distributed and with which arrays may be aligned.

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

42 3.2 Syntax of Data Alignment and Distribution Directives

44 Specification directives in HPF have two forms: specification statements, analogous to the
45 **DIMENSION** and **ALLOCATABLE** statements of Fortran 90; and an attribute form analogous to
46 type declaration statements in Fortran 90 using the “:” punctuation.

47 The attribute form allows more than one attribute to be described in a single directive.
48 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 64×64 templates; C is 32×32 .

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

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

3 Constraint: If either the *dist-format-clause* or the *dist-target* in a DISTRIBUTE directive
4 begins with “*” then every *distributee* must be a dummy argument.
5

6 Constraint: Neither the *dist-format-clause* nor the *dist-target* in a REDISTRIBUTE may begin
7 with “*”.
8

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

11 Note that the possibility of a DISTRIBUTE directive of the form

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

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

14 Examples:

```
15 !HPF$ DISTRIBUTE D1(BLOCK)
```

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

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

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

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

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

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

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

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

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

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

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

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):

12
13
14
15
16

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

results in this mapping of array elements onto abstract processors:

17
18
19
20
21
22
23
24
25
26
27
28
29
30

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,

31
32
33
34
35
36
37
38
39
40

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

is equivalent to

41
42

```
!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

43
44
45
46
47
48

```
!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 “:” 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 “:” 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 *
```

1	H323	<i>align-subscript-use</i>	is	[[<i>int-level-two-expr</i>] <i>add-op</i>] <i>align-add-operand</i>
2			or	<i>align-subscript-use add-op int-add-operand</i>
3	H324	<i>align-add-operand</i>	is	[<i>int-add-operand</i> *] <i>align-primary</i>
4			or	<i>align-add-operand * int-mult-operand</i>
5				
6	H325	<i>align-primary</i>	is	<i>align-dummy</i>
7			or	(<i>align-subscript-use</i>)
8	H326	<i>int-add-operand</i>	is	<i>add-operand</i>
9				
10	H327	<i>int-mult-operand</i>	is	<i>mult-operand</i>
11	H328	<i>int-level-two-expr</i>	is	<i>level-2-expr</i>

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

16 Constraint: The *align-spec* in a REALIGN may not begin with “*”.

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

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

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

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

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

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

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

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

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

46	J+J	J-J	3*K-2*K	M*(N-M)	2*J-3*J+J	2*(3*(K-1)+13)-K
47	J*J	J+K	3/K	2**M	M*K	K-3*M
48	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,

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

2
3 means that a copy of A is aligned with every column of D, because it is conceptually
4 equivalent to

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

6
7 just as

8
9 !HPF\$ ALIGN A(:,*) WITH D(:)

10
11 is conceptually equivalent to

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

13
14 Note, however, that while HPF syntax allows

15
16 !HPF\$ ALIGN A(:,*) WITH D(:)

17
18 to be written in the alternate form

19
20 !HPF\$ ALIGN A(:,J) WITH D(:)

21
22 it does *not* allow

23
24 !HPF\$ ALIGN A(:) WITH D(:,*)

25
26 to be written in the alternate form

27
28 !HPF\$ ALIGN A(:) WITH D(:,J)

29
30 because that has another meaning (only a variable appearing in the *align-source-list*
31 following the *alignee* is understood to be an *align-dummy*, so the current value of the
32 variable J is used, thus aligning A with a single column of D).
33

34 Replication allows an optimizing compiler to arrange to read whichever copy is closest.
35 (Of course, when a replicated data object is written, all copies must be updated, not
36 just one copy. Replicated representations are very useful for use as small lookup
37 tables, where it is much faster to have a copy in each physical processor but without
38 giving it an extra dimension that is logically unnecessary to the algorithm.) (*End of*
39 *rationale.*)
40

41 By applying the transformations given above, all cases of an *align-subscript* may be
42 conceptually reduced to either an *int-expr* (not involving an *align-dummy*) or an *align-*
43 *subscript-use* and the *align-source-list* may be reduced to a list of index variables with no “*”
44 or “:”. An *align-subscript-list* may then be evaluated for any specific combination of values
45 for the *align-dummy* variables simply by evaluating each *align-subscript* as an expression.
46 The resulting subscript values must be legitimate subscripts for the *align-target*. (This
47 implies that the *alignee* is not allowed to “wrap around” or “extend past the edges” of an
48 *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.

```

```

1      !Reversing both axes
2      !HPF$ ALIGN X(J,K) WITH D2(M-J+1,N-K+1)
3      !HPF$ ALIGN X(:, :) WITH D2(M:1:-1,N:1:-1)
4
5      !Simple case
6      !HPF$ ALIGN X(J,K) WITH D2(J,K)
7      !HPF$ ALIGN X(:, :) WITH D2(:, :)
8      !HPF$ ALIGN (J,K) WITH D2(J,K):: X
9      !HPF$ ALIGN (:, :) WITH D2(:,):: X
10     !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:

```

32     !HPF$ DYNAMIC A,B,C,D,E
33     !HPF$ DYNAMIC:: A,B,C,D,E
34     !HPF$ DYNAMIC, ALIGN WITH SNEEZY:: X,Y,Z
35     !HPF$ ALIGN WITH SNEEZY, DYNAMIC:: X,Y,Z
36     !HPF$ DYNAMIC, DISTRIBUTE(BLOCK, BLOCK) :: X,Y
37     !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

```

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

```

may be combined into a single directive as follows:

```

47     !HPF$ TEMPLATE, DISTRIBUTE(BLOCK, BLOCK) ONTO P,    &
48     !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

1 allocated; however, the first `ALLOCATE` statement is nonconforming because `S` needs to be
 2 aligned but at that point in time `T` is still unallocated.

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

```

  9         REAL,ALLOCATABLE(:,:) :: TINKER, EVERS
 10 !HPF$ DYNAMIC :: TINKER, EVERS
 11         REAL, POINTER :: CHANCE(:)
 12 !HPF$ DISTRIBUTE(BLOCK),DYNAMIC :: CHANCE
 13         ...
 14         READ 6,M,N
 15         ALLOCATE(TINKER(N*M,N*M))
 16 !HPF$ REDISTRIBUTE TINKER(CYCLIC, BLOCK)
 17         ALLOCATE(EVERS(N,N))
 18 !HPF$ REALIGN EVERS(:,) WITH TINKER(M:M,1:M)
 19         ALLOCATE(CHANCE(10000))
 20 !HPF$ REDISTRIBUTE CHANCE(CYCLIC)
 21
  
```

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

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

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

36
 37 *Advice to implementors.* If HPF directives were ever to be absorbed as actual
 38 Fortran statements, the previous paragraph could be written as “Remapping an object
 39 causes all pointers associated with any portion of the object to have undefined pointer
 40 association status, except for the one pointer, if any, used to indicate the object in
 41 the remapping directive.” The more complicated wording here is intended to avoid
 42 any implication that the remapping directives, in the form of structured comment
 43 annotations, have any effect on the execution semantics, as opposed to the execution
 44 speed, of the annotated program.) (*End of advice to implementors.*)

45
 46 When an array is allocated, it will be aligned to an existing template if there is an
 47 explicit `ALIGN` directive for the allocatable variable. If there is no explicit `ALIGN` directive,
 48 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

1 replicated over all processors. For target architectures that have a set of computational
 2 processors and a separate scalar host computer, a natural implementation is to map
 3 every scalar processor arrangement onto the host processor. For target architectures
 4 that have a set of computational processors but no separate scalar “host” computer,
 5 data mapped to a scalar processor arrangement might be mapped to some arbitrarily
 6 chosen computational processor or replicated onto all computational processors. (*End*
 7 *of rationale.*)

8
 9 An HPF compiler is required to accept any PROCESSORS declaration in which the prod-
 10 uct of the extents of each declared processor arrangement is equal to the number of physical
 11 processors that would be returned by the call NUMBER_OF_PROCESSORS(). It must also accept
 12 all declarations of scalar PROCESSOR arrangements. Other cases may be handled as well,
 13 depending on the implementation.

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

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

18
 19 As in Fortran 90, an *explicit-shape-spec-list* in a *processors-decl* will override an explicit
 20 DIMENSION attribute:

```
21
22 !HPF$ PROCESSORS, DIMENSION(3,3,3) ::      &
23 !HPF$          RUBIK, RUBIKS_REVENGE(4,4,4), SOMA
```

24
 25 Here RUBIKS_REVENGE is $4 \times 4 \times 4$ while RUBIK and SOMA are each $3 \times 3 \times 3$. (By the rules
 26 enunciated above, however, such a statement may not be completely portable because no
 27 HPF language processor is required to handle shapes of total sizes 27 and 64 simultaneously.)

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

- 32
- 33 • The array or template itself becomes undefined at the same time by virtue of returning
- 34 from the subprogram.
- 35
- 36 • Whenever the subprogram is called, the processor arrangement is always locally de-
- 37 fined in the same way, with identical lower bounds, and identical upper bounds.
- 38

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

42
 43 Variables in COMMON or having the SAVE attribute may be mapped to a locally declared
 44 processor arrangement, but because the first condition cannot hold for such variables (they
 45 don’t become undefined), the second condition must be observed. This allows COMMON
 46 variables to work properly through the customary strategy of putting identical declarations
 47 in each scoping unit that needs to use them, while allowing the processor arrangements to
 48 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)
!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.

```

1 H334 template-directive      is TEMPLATE template-decl-list
2 H335 template-decl          is template-name [ ( explicit-shape-spec-list ) ]
3
4 H336 template-name          is object-name

```

Examples:

```

7 !HPF$ TEMPLATE A(N)
8 !HPF$ TEMPLATE B(N,N) , C(N,2*N)
9 !HPF$ TEMPLATE DOPEY(100,100) ,SNEEZY(24) ,GRUMPY(17,3,5)
10

```

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.

```

16 !HPF$ TEMPLATE, DISTRIBUTE(BLOCK,*) :: &
17 !HPF$                               WHINEY(64,64) ,MOPEY(128,128)
18 !HPF$ TEMPLATE, DIMENSION(91,91) :: BORED,WHEEZY,PERKY
19

```

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 $(N + 1) \times (N + 1)$:

```

25 !HPF$ TEMPLATE, DISTRIBUTE(BLOCK, BLOCK) :: EARTH(N+1,N+1)
26 REAL, DIMENSION(N,N) :: NW, NE, SW, SE
27 !HPF$ ALIGN NW(I,J) WITH EARTH( I , J )
28 !HPF$ ALIGN NE(I,J) WITH EARTH( I ,J+1)
29 !HPF$ ALIGN SW(I,J) WITH EARTH(I+1, J )
30 !HPF$ ALIGN SE(I,J) WITH EARTH(I+1,J+1)
31

```

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)

```

1 !HPF\$ INHERIT BREAD

2
3
4 The inherited template of `BREAD` has shape `[100]`; element `BREAD(I)` is aligned with
5 element `5 + 2*I` of the inherited template and, since `BREAD` does not appear in a prescriptive
6 `DISTRIBUTE` directive, it has a `BLOCK(10)` distribution.

7 8 3.10 Alignment, Distribution, and Subprogram Interfaces

9
10 Mapping directives may be applied to dummy arguments in the same manner as for other
11 variables; such directives may also appear in interface blocks. However, there are addi-
12 tional options that may be used only with dummy arguments: asterisks, indicating that a
13 specification is descriptive rather than prescriptive, and the `INHERIT` attribute.

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

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

27
28 In order to describe explicitly the distribution of a dummy argument, the template
29 that is subject to distribution must be determined. A dummy argument always has a fresh
30 template to which it is ultimately aligned; this template is constructed in one of three ways:

- 31
- 32 • If the dummy argument appears explicitly as an *alignee* in an `ALIGN` directive, its
33 template is specified by the *align-target*.
 - 34 • If the dummy argument is not explicitly aligned and does not have the `INHERIT`
35 attribute, then the template has the same shape and bounds as the dummy argument;
36 this is called the *natural template* for the dummy.
 - 37 • If the dummy argument is not explicitly aligned and does have the `INHERIT` attribute,
38 then the template is “inherited” from the actual argument according to the following
39 rules:
40
 - 41 – If the actual argument is a whole array, the template of the dummy is a copy of
42 the template with which the actual argument is ultimately aligned.
 - 43 – If the actual argument is a regular array section of array *A*, then the template
44 of the dummy is a copy of the template with which *A* is ultimately aligned.
 - 45 – If the actual argument is any other expression, the shape and distribution of the
46 template may be chosen arbitrarily by the language processor (and therefore the
47 programmer cannot know anything *a priori* about its distribution).
- 48

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	1, 2, 3
2	2	4, 5, 6
3	3	7, 8
4	4	9, 10, 11
5	5	12, 13, 14
6	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	1			9												
2		4			12											
3			7													
4	2			10												
5		5			13											
6			8													
7	3			11												
8		6			14											

but we shall see that this is not permitted in HPF. Within subroutine TERPSICHOE 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.

1 !HPF\$ DISTRIBUTE POLYHYMNIA * ONTO ELVIS

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

5 !HPF\$ DISTRIBUTE THALIA *(CYCLIC) ONTO FLIP

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

9 !HPF\$ DISTRIBUTE CALLIOPE (CYCLIC) ONTO *HOMER

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

13 !HPF\$ DISTRIBUTE MELPOMENE * ONTO *EURIPIDES

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

17 !HPF\$ DISTRIBUTE CLIIO *(CYCLIC) ONTO *HERODOTUS

18 CLIIO is asserted to already be distributed CYCLIC onto HERODOTUS so, if possible, no data
19 movement should occur.

20 !HPF\$ DISTRIBUTE EUTERPE (CYCLIC) ONTO *

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

24 !HPF\$ DISTRIBUTE ERATO * ONTO *

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

27 !HPF\$ DISTRIBUTE ARTHUR_MURRAY *(CYCLIC) ONTO *

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

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

32 !HPF\$ DISTRIBUTE ERATO *(*) ONTO *

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

36 One may omit either the *dist-format-clause* or the *dist-target-clause* for a dummy ar-
37 gument. If such a clause is omitted and the dummy argument has the INHERIT attribute,
38 then the compiler must handle the directive as if * or ONTO * had been specified explicitly.
39 If such a clause is omitted and the dummy does not have the INHERIT attribute, then the
40 compiler may choose the distribution format or a target processor arrangement arbitrarily.
41 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

1 simply BLOCK, but there will be some n that describes the resulting distribution) and
 2 LB is the lower bound for that axis of the array, then $BLOCK(n/s)$ should appear in
 3 the distribution for the natural template, *provided* that s divides n evenly and that
 4 $l - LB < s$.

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

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

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

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

23 Each column of TROGGS has a BLOCK distribution. The rules listed above justify the pro-
 24 grammer in saying so:

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

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

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

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

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

41 In this example the first dummy argument, TSAR, is allowed to remain aligned with the
 42 corresponding actual argument, while the second dummy argument, CZAR, is forced to be
 43 aligned with the first dummy argument. If the two actual arguments are already aligned,
 44 no remapping of the data will be required at run time; but the subprogram will operate
 45 correctly even if the actual arguments are not already aligned, at the cost of remapping the
 46 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

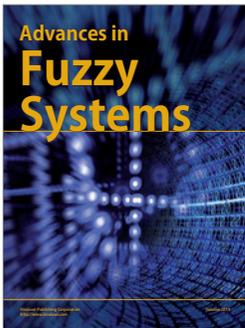
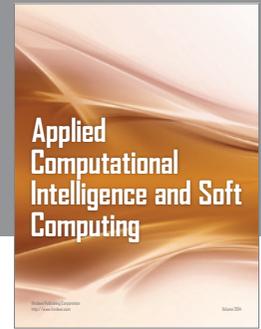
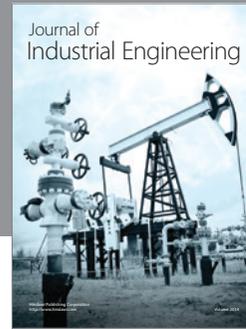
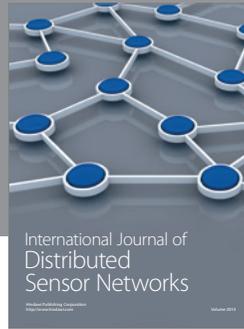
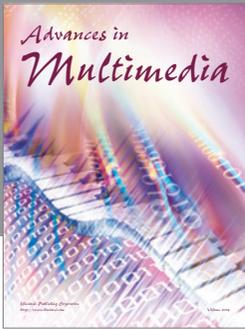
PROGRAM MAIN
```

```
1           USE FOO
2           CALL SUB(A(1:5,3:9))
3           END
4
5           SUBROUTINE SUB(B)
6           USE FOO
7           REAL B(:, :)
8           ...
9           !HPF$ REDISTRIBUTE A           !nonconforming
10          ...
11          END
12
```

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

18
19 An overriding principle is that any mapping or remapping of arguments is not visible
20 to the caller. This is true whether such remapping is implicit (in order to conform to
21 prescriptive directives, which may themselves be explicit or implicit) or explicit (specified
22 by **REALIGN** or **REDISTRIBUTE** directives). When the subprogram returns and the caller
23 resumes execution, all objects accessible to the caller after the call are mapped exactly as
24 they were before the call. It is not possible for a subprogram to change the mapping of any
25 object in a manner visible to its caller, not even by means of **REALIGN** and **REDISTRIBUTE**.

26
27 *Advice to implementors.* There are several implementation strategies for achieving
28 this behavior. For example, one may be able to use a copy-in/copy-out strategy for
29 arguments that require remapping on subprogram entry. Alternatively, one may be
30 able to remap the actual argument on entry and remap again on exit to restore the
31 original mapping. (*End of advice to implementors.*)
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48



Hindawi

Submit your manuscripts at
<http://www.hindawi.com>

