Description

Being specified in the `on` or the `from` clause of some directives, the template reference refers to a subset of a node set in which the specified subset of the template resides.

Specifically, the "`*`" symbol that appears as *template-subscript* in a dimension of *template-ref* is interpreted by each node at runtime as the indices of the elements in the dimension that reside in the node. "`*`" in a template reference is similar to "`*`" in a node reference.

Examples

Assume that `t` is a template.

- In the `task` directive, the executing node set of the task can be indirectly specified using a template reference in the `on` clause.

XcalableMP Fortran
```
!$xmp task on t(1:m,1:n)
!$xmp task on t
```
XcalableMP C
```
#pragma xmp task on t[0:n][0:m]
#pragma xmp task on t
```

- In the `loop` directive, the executing node set of each iteration of the following loop is indirectly specified using a template reference in the `on` clause.

XcalableMP Fortran
```
!$xmp loop (i) on t(i-1)
```
XcalableMP C
```
#pragma xmp loop (i) on t[i-1]
```

- In the `array` directive, the executing node set on which the associated array-assignment statement is performed in parallel is indirectly specified using a template reference in the `on` clause.

XcalableMP Fortran
```
!$xmp array on t(1:n)
```
XcalableMP C
```
#pragma xmp array on t[0:n]
```

- In the `barrier`, `reduction`, and `bcast` directives, the node set that is to perform the operation collectively can be indirectly specified using a template reference in the `on` clause.

XcalableMP Fortran
```
!$xmp barrier on t(1:n)
!$xmp reduction (+:a) on t(*,:)
!$xmp bcast (b) on t(1:n)
```
XcalableMP C
```
#pragma xmp barrier on t[0:n]
#pragma xmp reduction (+:a) on t[:][*]
#pragma xmp bcast (b) on t[0:n]
```

4.3.3 `distribute` Directive

Synopsis

The `distribute` directive specifies the distribution of a template.

Syntax

[F] `!$xmp distribute` *template-name* `(`*dist-format [*, *dist-format]...* `)` `onto` *nodes-name*

[C] `#pragma xmp distribute` *template-name* `(`*dist-format [*, *dist-format]...* `)` ▮
▮ `onto` *nodes-name*

[C] `#pragma xmp distribute` *template-name* `[` *dist-format* `]` *[* `[` *dist-format* `]` *...]* ▮
▮ `onto` *nodes-name*

where *dist-format* must be one of:

```
*
block [ ( int-expr ) ]
cyclic [ ( int-expr ) ]
gblock ( { * | int-array } )
```

Description

According to the specified distribution format, a template is distributed onto a specified node array. The dimension of the node array that appears in the `onto` clause corresponds, in order of left-to-right, to the dimension of the distributed template for which the corresponding *dist-format* is not "`*`".

Let `d` be the size of the dimension of the template, `p` be the size of the corresponding dimension of the node array, `ceiling` and `mod` be Fortran's intrinsic functions, and each of the arithmetic operators be that of Fortran. The interpretation of *dist-format* is as follows:

"`*`" The dimension is not distributed.

`block` Equivalent to `block(ceiling(d/p))`.

`block(n)` The dimension of the template is divided into contiguous blocks of size `n`, which are distributed onto the corresponding dimension of the node array. The dimension of the template is divided into `d/n` blocks of size `n`, and one block of size `mod(d,n)` if any, and each block is assigned sequentially to an index along the corresponding dimension of the node array. Note that if `k = p-d/n-1 > 0`, then there is no block assigned to the last `k` indices.

`cyclic` Equivalent to `cyclic(1)`.

`cyclic(n)` The dimension of the template is divided into contiguous blocks of size `n`, and these blocks are distributed onto the corresponding dimension of the node array in a round-robin manner.

`gblock(m)` `m` is referred to as a mapping array. The dimension of the template is divided into contiguous blocks so that the i'th block is of size `m(i)`, and these blocks are distributed onto the corresponding dimension of the node array.

If at least one `gblock(*)` is specified in *dist-format*, then the template is initially undefined and must not be referenced until the shape of the template is defined by `template_fix` directives at runtime.

Restrictions

- `[C]` *template-name* must be declared by a `template` directive that lexically precedes the directive.
- The number of *dist-format* that is not "`*`" must be equal to the rank of the node array specified by *nodes-name.*
- The size of the dimension of the template specified by *template-name* that is distributed by `block(n)` must be equal to or less than the product of the block size `n` and the size of the corresponding dimension of the node array specified by *nodes-name.*

- The array *int-array* in parentheses following `gblock` must be an integer one-dimensional array, and its size must be equal to the size of the corresponding dimension of the node array specified by *nodes-name*.
- Every element of the array *int-array* in parentheses following `gblock` must have a value of a nonnegative integer.
- The sum of the elements of the array *int-array* in parentheses following `gblock` must be equal to the size of the corresponding dimension of the template specified by *template-name*.
- `[C]` A `distribute` directive for a template must precede any of its references in the executable code in the block.
- A template can be distributed only once by a `distribute` directive.

Examples

Example 1

XcalableMP Fortran

```
!$xmp nodes p(4)
!$xmp template t(64)
!$xmp distribute t(block) onto p
```

XcalableMP C

```
#pragma xmp nodes p[4]
#pragma xmp template t[64]
#pragma xmp distribute t[block] onto p
```

The template `t` is distributed in `block` format, as shown in the following table.

p(1)	t(1:16)
p(2)	t(17:32)
p(3)	t(33:48)
p(4)	t(49:64)

p[0]	t[0:16]
p[1]	t[16:16]
p[2]	t[32:16]
p[3]	t[48:16]

Example 2

XcalableMP Fortran

```
!$xmp nodes p(4)
!$xmp template t(64)
!$xmp distribute t(cyclic(8)) onto p
```

XcalableMP C

```
#pragma xmp nodes p[4]
#pragma xmp template t[64]
#pragma xmp distribute t[cyclic(8)] onto p
```

The template `t` is distributed in `cyclic` format of size eight, as shown in the following table.

p(1)	t(1:8) t(33:40)
p(2)	t(9,16) t(41:48)
p(3)	t(17,24) t(49:56)
p(4)	t(25,32) t(57:64)

p[0]	t[0:8] t[32:8]
p[1]	t[8:8] t[40:8]
p[2]	t[16:8] t[48:8]
p[3]	t[24:8] t[56:8]

Example 3

XcalableMP Fortran

```
!$xmp nodes p(8,5)
!$xmp template t(64,64,64)
!$xmp distribute t(*,cyclic,block) onto p
```

XcalableMP C

```
#pragma xmp nodes p[5][8]
#pragma xmp template t[64][64][64]
#pragma xmp distribute t[block][cyclic][*] onto p
```

The first dimension of the template `t` is not distributed. The second dimension is distributed onto the first dimension of the node array `p` in `cyclic` format. The third dimension is distributed onto the second dimension of `p` in `block` format. The results are as follows:

`p(1,1)`	`t(1:64, 1:57:8, 1:13)`
`p(2,1)`	`t(1:64, 2:58:8, 1:13)`
...	...
`p(8,5)`	`t(1:64, 8:64:8, 53:64)`

`p[0][0]`	`t[0:13][0:8:8][0:64]`
`p[0][1]`	`t[0:13][1:8:8][0:64]`
...	...
`p[4][7]`	`t[52:12][7:8:8][0:64]`

Note that the “64” in `template t` is not divisible by “5” in `node p`. Thus, the sizes of the blocks are different among nodes.

4.3.4 `align` Directive

Synopsis

The `align` directive specifies that an array is to be mapped in the same way as a specified template.

Syntax

`[F]` `!$xmp align` *array-name* `(` *align-source [*, *align-source]...* `)` ▌
▌ `with` *template-name* `(`*align-subscript [*, *align-subscript]...* `)`

`[C]` `#pragma xmp align` *array-name* `[`*align-source*`]` *[*`[`*align-source*`]`*]...* ▌
▌ `with` *template-name* `(`*align-subscript [*, *align-subscript]...* `)`
or
▌ `with` *template-name* `[`*align-subscript* `]` *[* `[` *align-subscript* `]`*...]*

where *align-source* must be one of:

scalar-int-variable
`*`
`:`

and *align-subscript* must be one of:

scalar-int-variable [{ `+` | `-` } *int-expr]*
`*`
`:`

Note that the variable *scalar-int-variable* that appears in *align-source* is referred to as an “align dummy variable” and *int-expr* appearing in *align-subscript* as an “align offset.”

Description

The array specified by *array-name* is aligned with the template that is specified by *template-name* so that each element of the array indexed by the sequence of *align-sources* is aligned with the element of the template indexed by the sequence of *align-subscripts*, where *align-sources* and *align-subscripts* are interpreted as follows:

1. The first form of *align-source* and *align-subscript* represents an align dummy variable and an expression of it, respectively. The align dummy variable is considered to range over all valid index values in the corresponding dimension of the array.

2. The second form “`*`” of *align-source* and *align-subscript* represents a dummy variable (not an align dummy variable) that does not appear anywhere in the directive.

- The second form of *align-source* is said to "collapse" the corresponding dimension of the array. As a result, the index along the corresponding dimension does not affect the determination of the alignment.
- The second form of *align-subscript* is said to "replicate" the array. Each element of the array is replicated, and is aligned to all index values in the corresponding dimension of the template.

3. The third form of *align-source* and the matching *align-subscript* represents the same align dummy variable whose range spans all valid index values in the corresponding dimension of the array. The matching of colons ("`:`") in the sequence of *align-sources* and *align-subscripts* is determined as follows:

   - `[F]` Colons in the sequence of *align-sources* and those in the sequence of *align-subscripts* are matched in corresponding left-to-right order, where any *align-source* and *align-subscript* that is not a colon is ignored.
   - `[C]` Colons in the sequence of *align-sources* in right-to-left order, and those in the sequence of `(`*align-subscript*`)`'s in left-to-right order are matched, or those in the sequence of `[`*align-subscript*`]`'s in right-to-left order are matched, where any *align-source* and *align-subscript* that is not a colon is ignored.

In XcalableMP C, an `align` directive for a dummy argument can be placed either outside the function body (as in the old style of C) or in it (as in the ANSI style).

Restrictions

- `[C]` *array-name* must be declared by a declaration statement that lexically precedes the directive.
- An align dummy variable may appear at most once in the sequence of *align-sources*.
- An align dummy variable may appear at most once in the sequence of *align-subscripts*.
- An *align-subscript* may contain at most one occurrence of an align dummy variable.
- The *int-expr* in an *align-subscript* may not contain any occurrence of an align dummy variable.
- The sequence of *align-sources* must contain exactly as many colons as contained by the sequence of *align-subscripts*.
- `[F]` The array specified by *array-name* must not appear as an *equivalence-object* in an `equivalence` statement.
- `[C]` An `align` directive for an array must precede any of its appearances in the executable code in the block.
- `[F]` The array specified by *array-name* shall not be initially defined.
- `[C]` The array specified by *array-name* shall not be initialized through an *initializer*.
- An array can be aligned only once by an `align` directive.

Examples

Example 1

XcalableMP Fortran
```
!$xmp align a(i) with t(i)
```
XcalableMP C
```
#pragma xmp align a[i] with t[i]
```

In XcalableMP Fortran, the array element `a(i)` is aligned with the template element `t(i)`. In XcalableMP C, the array element `a[i]` is aligned with the template element `t[i]`. These are equivalent to the following codes.

XcalableMP Fortran
```
!$xmp align a(:) with t(:)
```
XcalableMP C
```
#pragma xmp align a[:] with t[:]
```

Example 2

XcalableMP Fortran
```
!$xmp align a(*,j) with t(j)
```
XcalableMP C
```
#pragma xmp align a[j][*] with t[j]
```

In XcalableMP Fortran, the subarray `a(:,j)` is aligned with the template element `t(j)`. Note that the first dimension of `a` is collapsed. In XcalableMP C, the subarray `a[j][:]` is aligned with the template element `t[j]`. Note that the second dimension of `a` is collapsed.

Example 3

XcalableMP Fortran
```
!$xmp align a(j) with t(*,j)
```
XcalableMP C
```
#pragma xmp align a[j] with t[j][*]
```

In XcalableMP Fortran, the array element `a(j)` is replicated and aligned with each template element of `t(:,j)`. In XcalableMP C, the array element `a[j]` is replicated and aligned with each template element of `t[j][:]`.

Example 4

XcalableMP Fortran
```
!$xmp template t(n1,n2)
real a(m1,m2)
!$xmp align a(*,j) with t(*,j)
```
XcalableMP C
```
#pragma xmp template t[n2][n1]
double a[m2][m1]
#pragma xmp align a[j][*] with t[j][*]
```

In XcalableMP Fortran, the subarray `a(:,j)` is aligned with each template element of `t(:,j)`. In XcalableMP C, the subarray `a[j][:]` is aligned with each template element of `t[j][:]`.

By replacing "`*`" of the array `a` and "`*`" of the template `t` with a dummy variable `i` and `k`, respectively, this alignment can be interpreted as the following mapping.

[F] $a(i,j) \to t(k,j) \mid (i,j,k) \in (1:n1,\ 1:n2,\ 1:m1)$
[C] $a[j][i] \to t[j][k] \mid (i,j,k) \in (0:n1,\ 0:n2,\ 0:m1)$

4.3.5 `shadow` Directive

Synopsis

The `shadow` directive allocates the shadow area for a distributed array.

Syntax

[F] `!$xmp shadow` *array-name* `(` *shadow-width [*, *shadow-width]...* `)`

[C] `#pragma xmp shadow` *array-name* `[`*shadow-width*`]` *[*`[`*shadow-width*`]`*]...*

where *shadow-width* must be one of:

int-expr
int-expr : *int-expr*
`*`

Description

The `shadow` directive specifies the width of the shadow area of an array specified by *array-name*, which is used to communicate the neighbor element of the block of the array. When *shadow-width* is of the form "*int-expr* : *int-expr*," the shadow area of the width specified by the first *int-expr* is added at the lower bound, and that specified by the second one is added at the upper bound in the dimension. When *shadow-width* is of the form *int-expr*, the shadow area of the same width specified is added at both the upper and lower bounds in the dimension. When *shadow-width* is of the form "`*`", the entire area of the array is allocated on each node, and the area that it does not own is regarded as a shadow. This type of shadow is sometimes referred to as a "full shadow."

Note that the shadow area of a multi-dimensional array includes "obliquely-neighboring" elements, which are owned by the node whose indices are different in more than one dimension, and that the shadow area can also be allocated at the global lower and upper bounds of an array.

The data stored in the storage area declared by the `shadow` directive is referred to as a *shadow object*. A shadow object represents an element of a distributed array, and corresponds to the data object that represents the same element as itself. The corresponding data object is referred to as the *reflection source* of the shadow object.

Restrictions

- `[C]` *array-name* must be declared by a declaration statement that lexically precedes the directive.
- The value specified by *shadow-width* must be a nonnegative integer.
- The number of *shadow-width* must be equal to the number of dimensions (or rank) of the array specified by *array-name*.
- `[C]` A `shadow` directive for an array must precede any of its appearances in the executable code in the block.
- A shadow area for a distributed array can be allocated only once by a `shadow` directive.