# HPF+
# An Extension of HPF for Advanced Applications*

**Siegfried Benkner, Erwin Laure, Hans Zima**
**Institute for Software Technology and Parallel Systems**
**University of Vienna**
**Liechtensteinstrasse 22, A-1090 Vienna, Austria**

Deliverable: TR 99-01
Date: January 1999
Status: final
Confidentiality: public
Responsible: University of Vienna

The **HPF+** Consortium:
AVL GesmbH - Austria
European Centre for Medium-Range Weather Forecasts (ECMWF) - England
Engineering System International SA (ESI) - France
NA Software Ltd (NAS) - England
University of Pavia - Italy
University of Vienna - Austria



*The work described in this paper was partially supported by the ESPRIT IV Long Term Research Project 21033 "HPF+" of the European Commission and by the Special Research Program SFB F011 "AURORA" of the Austrian Science Fund.

# Contents

# 1 Introduction

In this report we specify HPF+, a language supporting the execution of advanced applications on scalable parallel architectures. HPF+ is an extension of Fortran 90/95 and based upon Vienna Fortran [20], Vienna Fortran 90 [7] and High Performance Fortran (HPF-1) [13]. It was developed in the ESPRIT project *HPF+* (see *http://www.par.univie.ac.at/hpf+*, in close connection with the HPF-2 standardization effort [13, 14, 16, 18]. HPF+ serves as the base language for the priority research project "AURORA" of the Austrian Science Fund (*http://www.vcpc.univie.ac.at/aurora*).

In order to provide adequate support for complex industrial and academic applications, HPF+ extends the data distribution features of HPF-1 by introducing new mechanisms which are applicable to irregular data structures. The handling of processors has been clarified and generalized in order to enable the distribution of arrays of different rank to the same processors and to influence load balancing. The concept of independent loops has been extended in order to reflect the requirements of the benchmark codes which are characterized by highly irregular access patterns. This includes mechanisms to influence the distribution of loop iterations, to specify reduction operations and reusable communication schedules, and to avoid communication overheads caused by calls to procedures from within such loops.

A number of ill-defined concepts of HPF-1 have been redesigned or eliminated. In particular, the data distribution model and the argument transfer mechanisms have been significantly simplified without the loss of required functionality.

## 1.1 HPF-1 versus HPF-2

The HPF-2 definition effort started in 1995 in order to extend HPF-1 by new features that broaden the applicability of the language, and to provide clarification, interpretation and correction of the HPF-1 language standard. The HPF Forum produced a new language version (HPF 2.0 [14]) at the end of January 1997.

The HPF-2 Language Specification is structured as follows:

- HPF 2.0 (base language)
- Chapter on portable/efficient features (previously called Kernel-HPF)
- Annex on Definition of original HPF subset (subset HPF-1)
- Annex of Approved Language Extensions

The HPF-2 base language is essentially a subset of HPF-1. Almost all of the new features addressed during the HPF-2 effort (e.g. mapping to processor sections, general block and indirect distributions, derived types, etc.) and a few of the existing HPF 1.1 features that were not yet available in commercial HPF compilers by that time (e.g. dynamic redistribution) have been included only within the Annex of Approved Language Extensions, and are not contained in the base language of HPF 2.0. As a consequence, many of these features are not supported by current HPF-2 compilers.

## 1.2 Summary of HPF+ Features

The provision of non-standard and irregular distributions, parallel loops with indirect array accesses, and additional language support for work distribution, reduction operations, and reuse of communication schedules is fundamental for all HPF+ applications. Other features aim at correcting current drawbacks and deficiencies of HPF, e.g. the alignment concept, the handling of processors, and the transfer of distributed arguments at procedure boundaries. The following list summarizes the language features available in the final version of HPF+. All features that have been added, extended, or modified with respect to the HPF-2 base language are marked accordingly. New features that are also addressed in the Approved Extensions of HPF 2 are written in *italics*.

- Handling of processor arrays
  - Establishing relationships between processor arrays (new)
  - Defining and naming subsets and reshapes of processor arrays (new)
- Basic Data Distribution Mechanisms
  - `BLOCK(M), CYCLIC(M)` distributions
  - Alignment (modified)
  - Static Distributions and Dynamic Distributions
- Extension of Data Distribution Mechanisms
  - *distribution to processor subsets* (new)
  - *general block distributions* (new)
  - *multi-block distributions* (new)
  - *indirect distributions* (new)
  - *distribution of derived type components* (new)
  - *distribution of pointers* (new)
  - *distribution ranges for dynamically distributed objects* (new)
  - *shadow specifications* (new)
  - halo specifications (new)
- Independent Loops
  - new-clauses
  - *reduction clauses and reduction statements* (new)
  - on-clauses and distribute-clauses for explicit work distribution (new)
- Distribution of Dummy Arrays
  - Explicit distributions
  - Inherited distributions (modified)
  - *distribution ranges for dummy arrays* (modified)
- Other Features
  - support for communication schedule reuse (new)
  - purest procedures (new)
  - `INDEPENDENT` directive for array assignments (new)
  - *support for task parallelism*(new)

## 1.3 Comparison with HPF 2.0

In Table 1 a comparison of the HPF+ language features with those provided in the HPF-2 base language and the Approved Extensions of HPF-2 is given.

| LANGUAGE FEATURE | HPF+ | HPF 2.0 | Approved Extensions |
|---|---|---|---|
| **Processors** | | | |
| processor views | + | – | – |
| processor subsets | + | – | – |
| **Data Distribution Mechanisms** | | | |
| block distributions | + | + | + |
| cyclic distributions | + | + | + |
| total replication | + | + | + |
| *GEN_BLOCK* | + | – | + |
| *MULTI_BLOCK* | + | – | – |
| dimensional *INDIRECT* | + | – | + |
| multi-dimensional *INDIRECT* | + | – | – |
| templates | – | + | + |
| shadow specifications | + | – | + |
| halo specifications | + | – | + |
| distribution to processor sections | + | – | + |
| dynamic distributions | + | – | + |
| distributions of derived type components | + | – | + |
| distributions of pointers | + | – | + |
| **Procedure Interface** | | | |
| inherited distributions | + | + | + |
| distribution ranges | + | – | + |
| prescriptive mappings | + | + | + |
| descriptive mappings | – | + | + |
| transcriptive mappings | + | + | + |
| procedures returning distributions | + | – | + |
| **Other Features** | | | |
| work distribution for parallel loops | + | – | + |
| purest procedures | + | – | – |
| schedule reuse | + | – | – |
| extrinsic program units | + | + | + |
| task parallelism | + | - | + |

Table 1: A comparison of selected language features in HPF+, HPF 2 and the Approved Extensions of HPF 2. "+" and "-" respectively indicate the presence or absence of a language feature.

The structure of this report is as follows. Section 2 describes the handling of processor arrays. Section 3 describes a number of new data distribution mechanisms which are crucial in the context of irregular applications. Section 4 specifies extensions of the independent loop concept. Section 5 introduces new language features provided for the manipulation of communication schedules. Section 7 defines the semantics of argument transfer in HPF+. Finally, in Section 8 language extensions w.r.t. task parallelism are discussed. Note that we do not specify features of HPF+ that may be used with the same semantics as in HPF. This includes extrinsic program units and the HPF library routines.

# 2 Processor Arrays

Whereas in HPF the user may define multiple processor arrays, the language does not provide any mechanisms for defining an explicit relationship between processor arrays of different shape. Hence programmers are forced to use templates and alignments whenever arrays of different rank[1] should be mapped to the same set of processors. But even with templates and alignments a number of quite simple distributions cannot be specified in HPF as shown in Example 2.1.

To address these limitations, HPF+ enables the programmer to impose different *views* on the set of available processors by establishing an explicit relationship between processor arrays according to the Fortran *array element order*. This is achieved by an extension of the processors directive that allows constructing processor arrays from sections of previously defined ones. The Fortran intrinsic function `RESHAPE` may be used in this context to specify arbitrary reshapes and to transpose the dimensions of multi-dimensional processor arrays.

**Example 2.1: Distributing Arrays of Different Rank**

```
!HPF$ PROCESSORS R2(2,2)
!HPF+ PROCESSORS R1(4) = RESHAPE(R2)

REAL A(10,10),B(100)

!HPF$ DISTRIBUTE(BLOCK,BLOCK) ONTO R2 :: A
!HPF$ DISTRIBUTE(BLOCK)       ONTO R1 :: B
```

*In this example, the two-dimensional array* `A` *and the one-dimensional array* `B` *are distributed to the same set of four processors. Since the HPF+ processors declaration defines a relationship between processor arrays* `R1` *and* `R2`*, the above directives specify the following distribution:*

- $A(1:5,1:5)$, $B(1:25)$, *to processor* $R2(1,1)$,
- $A(6:10,1:5)$, $B(26:50)$, *to processor* $R2(2,1)$,
- $A(1:5,6:10)$, $B(51:75)$, *to processor* $R2(1,2)$,
- $A(6:10,6:10)$, $B(76:100)$, *to processor* $R2(2,2)$.

*In HPF, the distribution for A and B the same processors as outlined above cannot be expressed since the user cannot assume any relationship between processor arrays of different shape.* ■

The HPF+ extension of *processors directives* enables the programmer to

- establish a relationship between different processor arrays,
- to provide different views of a processor array, and
- to define and name subsets (sections) of processor arrays.

[1] In HPF the number of distributed dimensions of an array must match the rank of the processor array to which the distribution refers.

All these features are expressed by means of an *associate processors* specification using a syntax similar to the Fortran syntax for variable initialization.

**Syntax:**

| | | |
|---|---|---|
| *processors-directive* | **is** | `PROCESSORS` *processors-decl-list* |
| *processors-decl* | **is** | *processors-name* [ (*explicit-shape-spec-list*) ] [ `=` *associate-proc-ref* ] |
| *associate-proc-ref* | **is** | *associate-processors-section* |
| | **or** | `RESHAPE (` *associate-processors-section* [, *order* ] `)` |
| *associate-processors-section* | **is** | *associate-processors-name* [ ( *section-subscript-list* ) ] |
| *order* | **is** | [ `ORDER =`] *order-expr* |
| *order-expr* | **is** | *int-expr* |

Constraint: All *section-subscripts* and *order-expr* must be specification expressions.

Constraint: If *order-expr* is specified, it must be a rank one integer array expression of size equal to the rank of *associate-processors-section*. The value of *order-expr* must be a permutation of `(/1,2,...,n/)`, where `n` is the rank of *associate-processors-section*.

If a *section-subscript* is a vector subscript, all values must lie between `1` and the extent of the corresponding dimension of the processor array designated by *associate-processors-name* and there must not be any repeated values.

A processor array that is referenced in an *associate-proc-ref* array must either be accessed from the host, or be accessed from a module, or be declared in a preceding statement, or be declared to the left of its use in the same processors directive.

## 2.1 Relationship between Processor Arrays

HPF+ provides two mechanisms in order to establish a relationship between processor arrays. The first mechanism states that two processor arrays that have the same shape *implicitly* refer to the same set of processors.[2] Note that this is the only way provided in HPF to define a relationship between processor arrays.

In addition to this implicit relationship of conformable processor arrays, in HPF+ a relationship between different processor arrays may be defined *explicitly* by means of specifying an *associate-processors-ref* within a processors declaration as described next.

## 2.2 Processor Views and Processor Subsets

By providing an *associate-processors-ref* within the declaration of a processor array `R` an explicit relationship between `R` and the processor array (section) referenced in the *associate-processors-ref* is established according to the Fortran array element order. Within the declaration of a particular processor array an *associate-processors-ref* may either be a reference to a section of another processor

[2] This is only true for processor arrays for which no *associate-proc-ref* has been specified.

array, or a reference to the intrinsic function RESHAPE with a section of another processor array as argument. In the first case, the processor array section specified in the *associate-processors-ref* must have the same shape as the processor array to be declared. In the second case, the processor array section appearing as argument to RESHAPE must have the same size as the processor array to be declared.

Subsets of processor arrays may be referenced by means of the usual Fortran mechanisms for array sections. A processor subset is *regular* if none of the *section-subscripts* is a *vector-subscript*, otherwise it is *irregular*.

The intrinsic function RESHAPE may be used to specify that a processor array R is a reshape of another previously declared processor array (section) Q possibly with permuted dimensions. Correspondence between R and Q is established according to the Fortran array element order. The optional *order* argument may be used to specify arbitrary transpositions of a multidimensional processor array and thus it must be a permutation of (/1,2,...,n/), where n is the rank of Q. Examples of HPF+ processor declarations are shown in Example 2.2.

**Example 2.2: Processor Views and Processor Subgroups**

```
!HPF$ PROCESSORS R(8)
!HPF+ PROCESSORS R2(4,2) = RESHAPE(R)
!HPF+ PROCESSORS R_ODD(4) = R(::2), R_EVEN(4) = R(2::2)
!HPF+ PROCESSORS R2T(2,4) = RESHAPE(R2,(/2,1/))
```

*The second processors directive declares a processor array* R2 *which corresponds to the same processors as* R. *Correspondence is established by means of the Fortran array element order, i.e.* $\forall i, j, 1 \leq i \leq 4, 1 \leq j \leq 2$, R2(i,j) *corresponds to* R(i+(j-1)*4). *The third directive defines a processor array that represents all processors of* R *with an odd index, i.e.* R_ODD(i) *corresponds to* R(2*i-1), $\forall i, 1 \leq i \leq 4$, *and a processor array* R_EVEN *representing all processors of* R *with an even index, respectively. In the last directive* R2T *is obtained by transposing* R2. *Thus processor* R2T(i,j) *corresponds to processor* R2(j,i), $\forall i, j, 1 \leq i \leq 2, 1 \leq j \leq 4$. ■

## 2.3 Size of Processor Arrays

According to [14] an HPF compiler is only required to support processor arrays which are either scalar or of size equal to NUMBER_OF_PROCESSORS(). Handling of other processor arrays in HPF is implementation dependent.

For consistency reasons and in order to ensure the portability of codes, in HPF+ the size of processor arrays may be equal *or less* than the number of physical processors that would be returned by the intrinsic function NUMBER_OF_PROCESSORS().

## 2.4 Discussion

In HPF the user may declare multiple processor arrays but has no means to specify an explicit relationship between different processor arrays. Only processor arrays with the same shape are guaranteed to refer to the same set of processors.

In Vienna Fortran [20] the user may declare multiple processor arrays which all must have the same size. All processor arrays refer to the same set of (physical) processors. A relationship between different processor arrays is established implicitly by means of the Fortran array element order [15]. Furthermore, an implicit one-dimensional processor array is provided.

In Vienna Fortran 90 [7] procedure local processor arrays may have a smaller size than the processor arrays of the main program.

The features provided by HPF+ are similar to those of Vienna Fortran but more powerful. HPF+ not only provides the user with mechanisms to establish an explicit relationship between different processor arrays, but also introduces features for defining and naming arbitrary subsets of processor arrays. This significantly improves the expressivity and flexibility of the data distribution mechanisms.

# 3 Data Distribution

There are a number of problems, in particular irregular applications that are based on unstructured meshes, for which the basic HPF distributions, `BLOCK` and `CYCLIC`, do not result in an adequate balance of the workload across the processors of the target machine. Furthermore, many codes from diverse application areas use multiple grids to model the underlying problem domain. Some of these applications exhibit different levels of parallelism; inter-grid parallelism if the computation on each grid may be performed independently of other grids, and intra-grid parallelism which is characterized by the loosely-synchronous data parallelism of structured grid codes. In order to exploit both levels of parallelism while providing the opportunity to balance the workload, it is necessary to distribute the grids to suitably sized subsets of processors.

To address the requirements of such applications HPF+ extends the basic data distribution mechanisms of HPF with:

- distribution to processor subsets
- general block distributions
- multi-block distributions
- indirect distributions

Before we describe these features in more detail, we discuss the data distribution model of HPF and HPF+.

## 3.1 Model

The data mapping model underlying HPF is a *two-level mapping.* First arrays have to be aligned with a template, and second, the template has to be distributed onto a processor array. The requirement that each array be aligned with a template (either explicitly or implicitly) significantly complicates the semantics of data mapping directives, in particular, when distributed arrays are transferred across procedure boundaries. Additional problems are encountered in the context of allocatable arrays since templates must not be dynamically allocated. Moreover, experience gained from porting large applications [17] to HPF indicates that templates and complex alignments are of very limited use when specifying data distributions. Therefore, without loss of required functionality, HPF+ simplifies the data distribution model by eliminating templates.

The fundamental data distribution model of HPF+ is a *one-level mapping* where arrays are distributed directly to virtual processor arrays or processor array sections. The number of distributed dimensions, $d$, of an array must be equal to or less than the rank, $r$, of the processor array to which the distribution refers. The distributed dimensions are mapped in left-to-right order to the first $d$ dimensions of the processor array, and, if $d < r$, data is replicated over the remaining $r - d$ processor dimensions. Alignment may be used to specify groups of arrays that are distributed in the same way. An alignment relationship between arrays is fixed and cannot be destroyed at runtime.

The establishment of a relationship between processor arrays of different shape and the provision of processor views as described in Section 2 further enhances the expressivity and flexibility of data distribution directives.

## 3.2 Basic Data Distribution Features

In the following we give an overview of the basic data distribution mechanisms of HPF+. A number of extensions to these mechanism are described in Sections 3.3 to 3.8.

In HPF+, an object is referred to as *explicitly distributed* if it is referenced in a `DISTRIBUTE` or an `ALIGN` directive, or, if it has the `DISTRIBUTE`, or the `ALIGN` attribute. An explicitly distributed object is called an *alignee* if it has the `ALIGN` attribute, otherwise it is called a *distributee*.

### 3.2.1 The `DISTRIBUTE` Directive

By means of the `DISTRIBUTE` directive or attribute the distribution of data objects to processor arrays may be specified. A `DISTRIBUTE` directive or attribute consists of the keyword `DISTRIBUTE` usually followed by a parenthesized list of distribution formats, one for each array dimension, and an optional `ONTO` clause which specifies the processor array to which the distribution refers. The basic distributions formats are `BLOCK` and `CYCLIC`, which indicate that the corresponding array dimension should be partitioned in a block or cyclic manner, or "*", which indicates that the dimension should not be distributed.

The syntax of the HPF+ `DISTRIBUTE` directive showing only the basic data distribution mechanisms is as follows. The complete specification of the syntax can be found Appendix.

| | | |
|---|---|---|
| *distribute-directive* | **is** | `DISTRIBUTE` *distributee dist-attribute-stuff* |
| *dist-attribute-stuff* | **is** | *dist-format-clause* [ *dist-onto-clause* ] |
| *distributee* | **is** | *object-name* |
| | **or** | ... |
| *dist-format-clause* | **is** | ( *dist-format-list* ) |
| | **or** | ( *generalized-dist-format* ) |
| *dist-format* | **is** | `BLOCK` [ ( *int-expr* ) ] |
| | **or** | `CYCLIC` [ ( *int-expr* ) ] |
| | **or** | `*` |
| | **or** | *extended-dist-format* |
| *dist-onto-clause* | **is** | `ONTO` *dist-target* |
| *dist-target* | **is** | *processors-name* |
| | **or** | *extended-dist-target* |

Constraint: The length of the *dist-format-list* must be equal to the rank of each distributee.

Constraint: If both a *dist-format-clause* and a *dist-onto-clause* appear, the number of elements of the *dist-format-list* that are not "`*`" must be equal to or less than the rank of the named processor arrangement of the *dist-onto-clause*.

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

The number of distribution formats of an array that are not "*" is referred to as its *distribution rank*. Whereas, in HPF the distribution rank of an array must always match the rank of the corresponding processor array, in HPF+ the distribution rank of an array must be equal to or less than the rank of the specified processor array. In the former case, the HPF+ distribution directive has exactly the same semantics as in HPF: the dimensions of the data array that are not "*" are distributed according to the specified distribution format in left-to-right order to the dimensions of the processor array. However, in the latter case, when the distribution rank of an array is less than the rank of the corresponding processor array, data is replicated over the remaining processor array dimensions as shown in Example 3.1.

### 3.2.2 Block and Cyclic Distributions

The basic data distribution mechanisms, e.g. `BLOCK` and `CYCLIC` distributions and the variants thereof with a block-size as argument, are supported in HPF+ in exactly the same way as in HPF. For sake of consistency these distribution formats are defined in the following.

**Definition: Block Distribution**

Let $A(1:N)$ denote a one-dimensional data array, $M \geq \left\lceil \frac{N}{P} \right\rceil$ an integer variable specifying the block-size and $R(1:P)$ a processor array. The block distribution

```
!HPF$ DISTRIBUTE A(BLOCK(M)) ONTO R
```

partitions the array into $b = \left\lfloor \frac{N}{P} \right\rfloor$ contiguous blocks of size $M$ which are distributed to the first $b$ processors, with $0 \leq b \leq$ P, and, if $N$ is not exactly divisible by $M$, one additional block with the remaining $(N \ mod \ M)$ elements which is assigned to processor $b+1$. All other processors (if any) do not receive any elements. If the optional block-size $M$ is omitted, it is if it were present with $M = \left\lceil \frac{N}{P} \right\rceil$.

The *data distribution function* $\delta : \mathbf{I}^A \to \mathbf{J}^R$ from the data array index dimension $\mathbf{I}^A$ to the processor array index dimension $\mathbf{J}^R$ is given by:

$$\delta(i) = \left\lceil \frac{i}{M} \right\rceil, 1 \leq i \leq N$$

**Definition: Cyclic Distribution**

Let $A(1:N)$ denote a one-dimensional data array, $M \geq 1$ an integer variable specifying the block-size and $R(1:P)$ a processor array. The cyclic distribution

```
!HPF$ DISTRIBUTE A(CYCLIC(M)) ONTO R
```

partitions the array into chunks of size $M$ which are distributed in a round-robin fashion to the processors. If $M$ is omitted, it is as if it were present with $M = 1$.

The *data distribution function* $\delta : \mathbf{I}^A \to \mathbf{J}^R$ from the data array index dimension $\mathbf{I}^A$ to the processor array index dimension $\mathbf{J}^R$ is given by:

$$\delta(i) = \left(1 + \left\lfloor \frac{i-1}{M} \right\rfloor\right) \ mod \ P, 1 \leq i \leq N$$

**Example 3.1: Basic Data Distribution Features**

```
!HPF$ PROCESSORS R1D(4), R2D(2,2)

      REAL A(8,8), B(8,8)

!HPF$ DISTRIBUTE (*,BLOCK) ONTO R1D :: A
!HPF$ DISTRIBUTE (*,BLOCK) ONTO R2D :: B
```

*Both distribution directives partition the arrays into blocks of columns. In case of array* `A` *each of these blocks of columns is mapped to one processor of the processor array* `RD1`*, i.e.* `A(:,1:2)` *is mapped to processor* `R1D(1)`*,* `A(:,3:4)` *is mapped to processor* `R1D(2)`*,* `A(:,5:6)` *is mapped to processor* `R1D(3)`*, and* `A(:,7:8)` *is mapped to processor* `R1D(4)`*. Array* `B` *is partioned into 2 blocks of columns which are distributed over the first dimension of processor array* `R2D` *and replicated over the second dimension, i.e.* `A(:,1:4)` *is mapped to processor* `R2D(1,1)` *and* `R2D(1,2)`*, and* `A(:,5:8)` *is mapped to processor* `R2D(2,1)` *and* `R2D(2,2)`*. Note that a distribution like that of array* `B` *can be specified in HPF only by means of templates and alignments.* ■

### 3.2.3 Alignment

The analysis of application codes and experience gained by others [17] from porting large scientific applications to HPF has shown that the rich alignment mechanism of HPF is too complex for real codes. HPF+, therefore, simplifies the alignment concept of HPF without sacrifycing required functionality. In HPF+ alignment may be used to specify that an array or an array dimension is distributed in the same way as another array or array dimension. An alignment relationship between two arrays is fixed in the sense that it cannot be destroyed at runtime. Thus alignment may be used to specify groups of arrays that are always distributed in the same way, even in the context of dynamic data distribution.

An alignee is said to be *immediately aligned* with its align-target if the align-target is an alignee. If the align-target is a distributee, it is said to be *ultimately aligned* with its align-target.

HPF+ align directives have one of the following forms:

`!HPF$ ALIGN` *alignee* `WITH` *align-target*

`!HPF$ ALIGN WITH` *align-target* :: *alignee*

`!HPF$ ALIGN` *alignee* ( *align-source-list* ) `WITH` *align-target* ( *align-subscript-list* )

`!HPF$ ALIGN` ( *align-source-list* ) `WITH` *align-target* ( *align-subscript-list* ) `::` *alignee*

The syntax of the `ALIGN` directive and attribute is as follows:

| | | |
|---|---|---|
| *align-directive* | **is** | `ALIGN` *alignee align-attribute-stuff* |
| *align-attribute-stuff* | **is** | `(` *align-source-list* `)` `WITH` *align-target* `(` *align-subscript-list* `)` |
| | **or** | `WITH` *align-target* |
| *alignee* | **is** | *object-name* |
| | **or** | *component-name* |
| *align-target* | **is** | *object-name* |
| | **or** | *component-name* |
| *align-source* | **is** | *align-dummy* |
| | **or** | `*` |
| *align-dummy* | **is** | *scalar-int-variable* |
| *align-subscript* | **is** | *align-dummy* |
| | **or** | `*` |

Constraint: An *alignee* must not be a *distributee.*

Constraint: An *alignee* must not have the `INHERIT` attribute.

Constraint: An *align-target* must not have the `OPTIONAL` attribute.

Constraint: If the *align-source-list* is present, its length must be equal to the length of each *alignee* to which it applies.

Constraint: An *align-dummy* must be a named variable, that appears in exactly one dimension of both the *align-source-list* and the *align-subscript-list.*

Constraint: If the *align-target* has the `DYNAMIC` attribute, the *alignee* must also have the `DYNAMIC` attribute.

The forms of the `ALIGN` directive or attribute without an *align-source-list* and an *align-subscript-list* are provided to specify *perfect alignment* of conformable arrays. For example `ALIGN A WITH B` specifies that $A$ and $B$ are distributed in exactly the same way, by mapping corresponding elements to the same processors. Note that this requires that $A$ and $B$ have the same shape. The other forms with an *align-source-list* and an *align-subscript-list* are provided to specify alignment with respect to individual array dimensions.

Each *align-source* corresponds to one dimension of the alignee and is either an *align-dummy* variable or a "*". Similarly, each *align-subscript* corresponds to one dimension of the align-target and is either an *align-dummy* variable or a "*".

- If the *align-source* in a dimension $d$ of the alignee is an *align-dummy* variable, then the same *align-dummy* variable must also appear exactly once in a dimension $d'$ of the *align-subscript-list.* As a consequence, dimension $d$ of the alignee is perfectly aligned with dimension $d'$ of the align-target, i.e. the corresponding indices of the alignee and the align-target are aligned. Note that this requires that dimension $d$ of the alignee and dimension $d'$ of the align-target have the same size.

- If an *align-source* is a "*", then the corresponding dimension of the alignee is *collapsed* and will not be distributed.

- If an *align-subscript* is a "*", the alignee is replicated over the corresponding dimension of the align-target.

The alignment mechanisms of HPF+ are a subset of those provided by HPF supporting perfect alignments, either of whole arrays or of individual array dimensions, collapsing of individual dimensions of the *alignee*, transposed alignments, and replication over individual dimensions of the *align-target* in the same way as in HPF. This is shown in Example 3.2.

Alignment strides and offset or embedding smaller dimensions into larger dimensions, as provided by HPF, are not supported. Moreover, the use of colons in an *align-source* and the use of *subscript-triplets* as *align-subscripts* are not supported for the sake of avoiding redundant syntax.

**Example 3.2: HPF+ Alignment Directives**

```
! perfect alignment
!HPF$ ALIGN A2 WITH B2
!HPF$ ALIGN A2(I,J) WITH B2(I,J)

! transposed alignment
!HPF$ ALIGN A2(I,J) WITH B2(J,I)

! collapsing a dimension of the alignee
!HPF$ ALIGN A2(I,*) WITH B1(I)

! replication over a dimension of the align-target
!HPF$ ALIGN A1(I) WITH B2(I,*)
```

■

Unlike HPF, the alignment mechanisms of HPF+ have been designed in such a way that the distribution of an array that is defined by means of alignment can always be represented by means of a direct distribution, i.e. with the `DISTRIBUTE` or `REDISTRIBUTE` directive. If the distribution of an alignee `A` is specified by means of an alignment with an align-target `B` whose distribution is specified by means of `DISTRIBUTE`, then the distribution of `A` can be expressed by means of `DISTRIBUTE` where the *dist-format* in dimension $d$, $1 \leq d \leq rank(\mathtt{A})$, is either

- a "*", if the *align-source* in dimension $d$ of `A` is a "*", or
- the *dist-format* of dimension $d'$ of `B`, if the *align-source* in dimension $d$ of `A` is an *align-dummy* variable that occurs in dimension $d'$ of the *align-subscript-list* of `B`.

The *dist-target* of $A$ is the same as that of $B$, if no transposition of aligned dimensions occurs, i.e. the left-to-right order of *align-dummy* variables in the *align-source-list* is the same as in the *align-subscript-list*. In the case of a transposed alignment, the first $a$ dimensions of the *dist-target* of `B` have to be permuted by the permutation $p = \{p_1, \ldots, p_n\}$, which must be a permutation of $\{1, \ldots, a\}$, where $a$ is the total number of *align-dummy* variables. The permutation $p$ is determined as follows: assume that the *align-dummy* variables in the *align-source-list* are denoted from left to right by $I_i$, then for all $i, 1 \leq i \leq a, p_i$ is given by the position of $I_i$ in the list of *align-dummy* variables that is constructed in left-to-right order from the *align-subscript-list*. Permuting the dimensions of a processor array can be achieved as described in Section 2.2 and 3.3.

## 3.3 Distribution to Subsets of Processors

Within the HPF-2 base language, distributions may refer only to whole processor arrays. Limited forms of distributions to subsets of processors can be achieved only by using templates and alignments. However, this is a rather inflexible method, which requires precise knowledge of the data arrays, templates and processor arrays, and thus cannot be applied to dynamically allocated data objects.

In HPF+ distributions to subsets of processors may be specified either by using as *distribution target* a processor array that has been associated with a subset of another processor array as defined on page 5, or by directly specifying a regular processor array section within the `ONTO` clause.

In the same way as within a *processors directive*, the `RESHAPE` function may be used in the *dist-onto-clause* to specify that a distribution target is a reshape of a processor array (section). In case of vector-subscripts, all values of the vector must lie between 1 and the extent of the corresponding dimension of the processor array and there must not be any repeated values.

The syntax for specifying processor subsets within the `ONTO` clause of a `DISTRIBUTE` or `REDISTRIBUTE` directive in HPF+ is as follows.

| | | |
|---|---|---|
| *dist-onto-clause* | **is** | `ONTO` *dist-target* |
| *dist-target* | **is** | *processors-name* |
| | **or** | *extended-dist-target* |
| *extended-dist-target* | **is** | *processors-section* |
| | **or** | `RESHAPE (` *processors-section* [`,` *shape* ] [`,` *order* ] `)` |
| *processors-section* | **is** | *processors-name* [ `(` *section-subscript-list* `)` ] |
| *order* | **is** | [ `ORDER =` ] *order-expr* |
| *shape* | **is** | [ `SHAPE =` ] *shape-expr* |
| *order-expr* | **is** | *int-expr* |
| *shape-expr* | **is** | *int-expr* |

Constraint: In a *section-subscript-list*, the number of *section-subscripts* must be equal to the rank of *processors-name*.

Constraint: *order-expr* and *shape-expr*, if present in a `REDISTRIBUTE` directive, must be restricted expressions.

Constraint: *order-expr* and *shape-expr*, if present, must be one-dimensional integer expressions of size equal to the rank of *processors-section*.

The value of *order-expr* must be a permutation of `(/1,2,...,n/)`, where `n` is the rank of *processors-section*.

Example 3.3 shows how distributions to regular processor subsets may be defined in HPF+.

## 3.4 Extended Data Distribution Formats

In this section we focus on additional data distribution features required for advanced applications like those considered within the HPF+ or the AURORA project. HPF+ extends the basic distribution formats by providing *general block*, *multi-block*, and *indirect distributions*.

**Example 3.3: Distribution to Processor Subsets**

```
!HPF+ PROCESSORS R(16), R2(4,4), UPPER_LEFT_R2(2,2) = R2(1:2,1:2)

!HPF$ DISTRIBUTE(BLOCK)       ONTO R(:8)          :: A
!HPF$ DISTRIBUTE(BLOCK)       ONTO R(9:)          :: B
!HPF+ DISTRIBUTE(BLOCK,BLOCK) ONTO UPPER_LEFT_R2 :: C
```

*Array A is partitioned into 8 blocks which are distributed to the first 8 processors of* `R`*. B is partitioned into 8 blocks which are distributed to the last 8 processors of* `R`*. Array* `C` *is partitioned into 4 blocks which are distributed to the processors in the upper left quarter of* `R2`*. Note that the user may not assume any relationship between processor arrays* `R` *and* `R2`*. If this were required,* `RESHAPE` *should be used as in Example 2.2.* ∎

### 3.4.1 General Block Distributions

The `BLOCK` distribution partitions an array dimension into contiguous blocks of (almost) equal size. However, for many applications it is crucial to specify blocks of different size due to load balancing requirements. For this purpose *general block distributions*, which partition an array dimension into arbitrarily sized contiguous blocks, are provided. Moreover, general block distributions provide enough flexibility to meet the demands of some irregular computations: if, for instance, the nodes of a simple unstructured mesh are partitioned prior to execution and then appropriately renumbered, the resulting distribution can be described in this manner.

With a general block distribution for each processor an individual block-size may be specified by using a one-dimensional integer array as an argument. The size of the argument array must match the number of processors to which the corresponding array (dimension) is distributed. The HPF+ general block distribution is defined as follows:

#### Definition: General Block Distribution

Let `A(1:N)` denote a one-dimensional data array, `B(1:P)` a one-dimensional integer array and `R(1:P)` a processor array. The general block distribution

```
!HPF$ DISTRIBUTE A(GEN_BLOCK(B)) ONTO R
```

partitions array `A` into `P` blocks. Block $i$ of size `B(i)` is mapped to processor `R(i)`, $\forall i, 1 \leq i \leq$`P`. Note that each element of $B$ must be non-negative and `SUM(B) = N` must be satisfied.

The *data distribution function* $\delta : \mathbf{I}^A \rightarrow \mathbf{J}^R$ from the data array index dimension $\mathbf{I}^A$ to the processor array index dimension $\mathbf{J}^R$ is given by:

$$\delta(i) = k \ s.t. \ \mathtt{SUM}(B(1:k-1)) < i \leq \mathtt{SUM}(B(1:k)), 1 \leq k \leq P, \ \forall i, 1 \leq i \leq P$$

The syntax of general block distributions and examples are shown on the next page.

**Syntax: General Block Distribution**

*extended-dist-format* **is** GEN_BLOCK ( *int-array-expr* )
**or** ...

Constraint: Any *int-array-expr* appearing as an argument to GEN_BLOCK in the *dist-format* of a DISTRIBUTE directive must be a *specification expression.*

Constraint: The *int-array-expr* appearing as an argument to GEN_BLOCK must be of rank 1 and of size equal to the rank of the corresponding dimension of the target processor arrangement.

Constraint: The sum of the values of the array obtained by evaluating *int-array-expr* of the GEN_BLOCK *dist-format* must be equal to the size of the dimension of the array being distributed. All values must be non-negative.

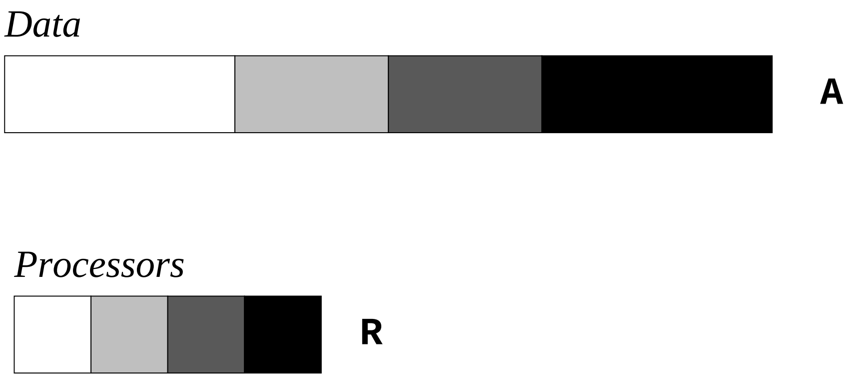
**Example 3.4: General Block Distribution**

```
!HPF$ PROCESSORS R(4)
      INTEGER, PARAMETER   :: BS(4) = (/ 30,20,20,30 /)
      REAL, DIMENSION(100) :: A, B

!HPF$ DISTRIBUTE A(GEN_BLOCK(BS)) ONTO R
!HPF$ DYNAMIC :: B

      INTEGER, DIMENSION(:), EXTERNAL :: COMPUTE_BLOCK_SIZE
      ...

!HPF$ REDISTRIBUTE (GEN_BLOCK(COMPUTE_BLOCK_SIZE())) ONTO R :: B
```




*In this example, Array A is distributed into 4 blocks, and block i of size* BS(i) *is mapped to processor* $R(i)$, $1 \leq i \leq 4$. B*, which is a dynamically distributed array (see Section 3.6), is redistributed by using the array-valued function* COMPUTE_BLOCK_SIZE() *as the argument of* GEN_BLOCK*.* ■

### 3.4.2 Multi-Block Distributions

Multi-block distributions are a combination of general block and indirect distributions. These distributions offer the flexibility of indirect distributions (see Section 3.4.3) while exploiting the structural regularity of general block distributions. Opposed to the general block distribution, which maps contiguous blocks of an array dimension one-to-one to the processors, multi-block distributions allow to assign contiguous blocks according to a user-defined, many-to-one relationship to the processors, and thus multiple blocks may be mapped to each processor. The `MULTI_BLOCK` distribution format requires two one-dimensional integer arrays (or array expressions) as arguments; the first argument specifies the size of each block, and the second argument specifies for each block the processor (index) to which it is distributed.

**Definition: Multi-Block Distribution**

Let `A(1:N)` denote a one-dimensional data array, `MB(1:P)` and `MM(1:P)` one-dimensional integer arrays, and `R(1:P)` a processor array. The multi-block block distribution

```
!HPF$ DISTRIBUTE A(MULTI_BLOCK(MB,MM)) ONTO R
```

partitions array `A` into `P` blocks. Block $i$ of size `MB(i)` is mapped to processor `R(MM(i))`, $\forall i, 1 \leq i \leq$`P`.

**Syntax: Multi-Block Distributions**

| | | |
|---|---|---|
| *extended-dist-format* | **is** | ... |
| | **or** | `MULTI_BLOCK` ( *int-array-expr* , *int-array-expr* ) |

Constraint: Any *int-array-expr* appearing as an argument to `MULTI_BLOCK` in the *dist-format* of a `DISTRIBUTE` directive must be a *specification expression.*

Constraint: Any *int-array-expr* appearing as an argument to `MULTI_BLOCK` must be of rank 1 and of size equal to the rank of the corresponding dimension of the target processor arrangement.

The sum of the values of the array which is obtained by evaluating the first *int-array-expr* expression of the `MULTI_BLOCK` distribution format must be equal to the size of the dimension of the array being distributed. All values must be non-negative.

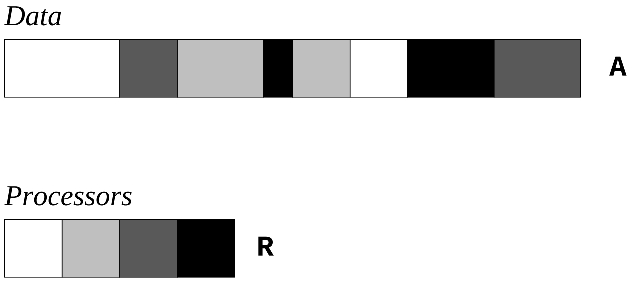
The values of the second *int-array-expr* of the `MULTI_BLOCK` distribution format must lie between 1 and size of the corresponding target processor array dimension.

A multi-block distribution is shown in Example 3.5.

### 3.4.3 Indirect Distributions

Indirect distributions, which are usually defined by means of an indirection array that specifies for each array element the processor to which it should be mapped, offer the ability to express totally unstructured or irregular distributions that do not involve replication. This flexibility however, requires

**Example 3.5: Multi-Block Distribution**




```
!HPF$ PROCESSORS R(4)
      INTEGER, PARAMETER   :: MB(8) = (/ 20,10,15,5,10,10,15,15 /)
      INTEGER, PARAMETER   :: MM(8) = (/ 1, 3, 2, 4, 2, 1, 4, 3 /)
      REAL, DIMENSION(100) :: A

!HPF$ DISTRIBUTE A(MULTI_BLOCK(MB,MM)) ONTO R
```

*In this example the data array* `A` *is partitioned into 8 blocks according to the contents of array* `MB`*. These blocks are distributed to the processors according to the contents of array* `MM`*. More precisely, block $i$ of size* `MB(`$i$`)` *is mapped to processor* `R(MM(`$i$`))`, $1 \leq i \leq 8$. ■

a different compilation strategy compared to regular distributions and sophisticated run time analysis and optimization techniques. Indirect distributions are usually defined by means of the `INDIRECT` distribution format taking a mapping array as an argument. The *mapping array* must be an integer array that is conformable with the array (dimension) to be distributed. A mapping array may itself be distributed in the usual way. If the mapping array is an array variable, changing its value later does *not* change the distribution.

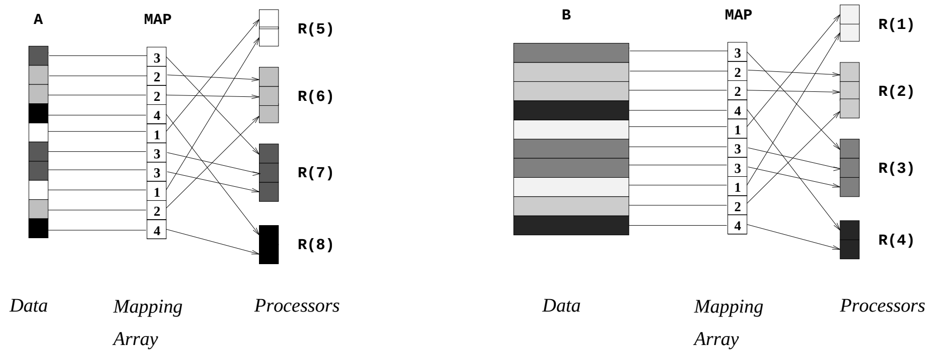
HPF+ provides two forms of indirect distributions - *dimensional indirect distributions* and *multidimensional indirect distributions*. Whereas a dimensional indirect distribution is restricted to a single array dimension, a multidimensional indirect distribution allows the user to specify an indirect distribution for a multidimensional array using a single conformable mapping array.

#### 3.4.3.1 Dimensional Indirect Distributions

A dimensional indirect distribution is used to specify an arbitrary replication-free distribution of a single array dimension to a single processor array dimension. In this case the mapping array must be 1-dimensional and must have the same size as the array dimension to be distributed. At the time the indirect distribution is evaluated, the value of each element of the mapping array must lie between `1` and the size of the corresponding processor array (dimension) to which the distribution refers.

By using array expressions, including array-valued functions, as argument of `INDIRECT`, the explicit use of mapping arrays can be avoided. Array valued functions referenced within the `INDIRECT` distribution format must be side-effect free (i.e. *pure*).

**Example 3.6: Dimensional Indirect Distribution**




```
      INTEGER, PARAMETER :: N=10, MAP(N)=(/3,2,2,4,1,3,3,1,2,4/), NP=8
      REAL, DIMENSION(N) :: A, B(N,N), C

!HPF$ PROCESSORS :: R(NP)
!HPF$ DISTRIBUTE (INDIRECT(MAP)) ONTO R(5:8)              :: A
!HPF$ DISTRIBUTE (INDIRECT(MAP),:) ONTO R(1:4), DYNAMIC :: B
!HPF$ DYNAMIC                                              :: C
      ....

!HPF$ REDISTRIBUTE (INDIRECT(MAP_FUNCTION(N,NP))) ONTO R :: C
```

*In this example element* `A(i)` *is mapped to processor* `R(MAP(i)+4)` *and the i-th row of* `B`*, i.e.* `B(i,:)`*, is mapped to processor* `R(MAP(i))`*,* $\forall i, 1 \leq i \leq 4$*. The dynamically distributed array* `C` *is distributed indirectly by using the array-valued function* `MAP_FUNCTION`*.* ■

### Definition: Dimensional Indirect Distribution

Let *A(1:N)* denote a one-dimensional data array, *MAP(1:N)* a one-dimensional integer array and *R(1:P)* a processor array. The indirect distribution

```
!HPF$ DISTRIBUTE A(INDIRECT(MAP)) ONTO R
```

distributes the array element $A(i)$ to processor $R(MAP(i))$, forall $1 \leq i \leq N$.

The *data distribution function* $\delta : \mathbf{I}^A \to \mathbf{J}^R$ from the data array index dimension $\mathbf{I}^A$ to the processor array index dimension $\mathbf{J}^R$ is given by:

$$\delta(i) = \mathtt{MAP}(i),\ 1 \leq i \leq N$$

Note that in the case where the lower bound of either the data array dimension, the mapping array or the processor array dimension is not 1 the $i^{th}$ element of the data array dimension is mapped to the $j^{th}$ element of the processor array dimension to which the distribution refers, where $j$ is given by the value of the $i^{th}$ element of the mapping array, and $\forall i, 1 \leq i \leq N$, and $1 \leq j \leq P$.

**Syntax: Dimensional Indirect Distributions**

*extended-dist-format* **is** ...
**or** INDIRECT (*int-array-expr*)

Constraint: Any *int-array-expr* appearing as an argument to INDIRECT in a *dist-format* must be a integer array expression of rank 1.

Constraint: The size of *int-array-expr* appearing as an argument to INDIRECT in a *dist-format* must be equal to the extent of the corresponding array dimension being distributed.

Constraint: The *int-array-expr* appearing as an argument to INDIRECT in the *dist-format* of a DISTRIBUTE directive must be a specification expression.

The values of *int-array-expr* must lie between 1 and size of the corresponding target processor array dimension.

Note that INDIRECT distributions as specified in the Approved Extensions of HPF-2 have a slightly different semantics: the values of the mapping array must lie between the declared lower-bound and upper-bound of the processor array dimension. In HPF+, the values of the mapping array must lie between 1 and the size of the corresponding processor array dimension. This modification was necessary for a consistent definition of dimensional and multidimensional indirect distribution. Furthermore, it alleviates the specification of indirect distributions to processor array sections.

#### 3.4.3.2 Multi-Dimensional Indirect Distributions

The above definition of indirect distributions excludes the use of multidimensional mapping arrays, since the INDIRECT distribution format is restricted to a single array dimension. As in Vienna Fortran 90 [7] indirect distributions for multidimensional arrays may not only be specified in a dimension-wise fashion but also for whole arrays using a single multidimensional mapping array. This provides additional functionality which cannot be achieved with dimensional indirect distributions.

By means of a multidimensional indirect distribution it is possible to specify an arbitrary distribution (without replication) from an $n$-dimensional data array A to an $m$-dimensional processor array R. The mapping array MAP and the data array A to be distributed must be conformable. The value of each element of the mapping array must lie between 1 and the size of the corresponding processor array (section) to which the distribution refers. Note that HPF provides no support for multidimensional indirect distributions.

**Definition: Multi-Dimensional Indirect Distribution**

Assume that the size of the array A to be distributed and the size of the mapping array MAP are given by $N$, and the size of the corresponding processor array is given by $P$. Then the $i^{th}$ element of A is mapped to the $j^{th}$ element of the processor array to which the distribution refers[3], where $j$ is given by the value of the $i^{th}$ element of the mapping array, and $\forall i, 1 \leq i \leq N$, and $1 \leq j \leq P$.

[3]This is based on the Fortran array element order.

Note that in case of a multidimensional indirect distribution the HPF restriction that the number of distributed dimensions must be equal to the rank of the processor array to which the distribution refers does not apply.

**Syntax: HPF+ Multi-Dimensional Indirect Distributions**

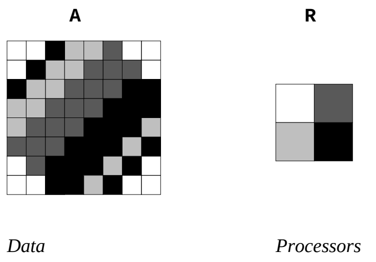
| | | |
|---|---|---|
| *dist-format-clause* | **is** | ( *dist-format-list* ) |
| | **or** | ( *generalized-dist-format* ) |
| *generalized-dist-format* | **is** | INDIRECT ( *int-array-expr* ) |

Constraint: Any *int-array-expr* appearing as an argument to INDIRECT in a *generalized-dist-format* must have the same shape as the array being distributed.

Constraint: The *int-array-expr* appearing as an argument to INDIRECT in the *generalized-dist-format* of a DISTRIBUTE directive must be a specification expression.

The values of *int-array-expr* must lie between 1 and the size of the corresponding target processor array (section).

**Example 3.7: Multi-Dimensional Indirect Distributions**




```
!HPF$ PROCESSORS R(2,2)
      DIMENSION A(8,8)
      INTEGER MAP(8,8)=RESHAPE((/1,1,4,2,2,3,1,1,    &
                                1,4,2,2,3,3,3,1,    &
                                4,2,2,3,3,3,4,4,    &
                                2,2,3,3,3,4,4,4,    &
                                2,3,3,3,4,4,4,2,    &
                                3,3,3,4,4,4,2,4,    &
                                1,3,4,4,4,2,4,1,    &
                                1,1,4,4,2,4,1,1 /), &
                                SHAPE=(/8,8/) )

!HPF$ DISTRIBUTE A(INDIRECT(MAP)) ONTO R
```

*To specify the shown indirect distribution a multi-dimensional mapping array is needed.* ∎

## 3.5 Non-Local Access Specifications

A major task of an HPF compiler, when transforming executable constructs that reference distributed arrays, is to determine non-local data accesses and to generate the necessary communication. In order to detect non-local data references, the access patterns, the ownership of the accessed data, and the processors that perform the accesses must be known. If any of this information cannot be derived statically by the compiler, for example if arrays are accessed indirectly, communication generation has to be deferred to the runtime. Despite a potentially large preprocessing overhead, communication generation based on runtime techniques may inhibit a number of crucial optimizations with respect to the management of non-local data and the related index conversion techniques.

HPF+, therefore, allows the user to provide the compiler or the runtime system with high level information about non-local accesses without requiring to deal with the details of message passing. The (super) set of non-local elements of a distributed array that may be accessed during the execution of the program may be specified by the user by defining *shadow regions* or *halo regions*. A shadow region is a regular, contiguous area around the local section of an array that is used to store copies of non-local data elements accessed on a particular processor. In contrast, a halo region is an irregular, non-contiguous area which usually has a different size on each processor.

The information provided by shadow or halo regions is utilized by the compiler to allocate memory (buffers) for all non-local accesses and to establish a corresponding address translation scheme. Furthermore, the use of such regions may significantly improve the organization of communication in the context of parallel loops.

### 3.5.1 Shadow Specifications

A shadow region is a rectilinear, contiguous area around the local section of an array used for storing non-local data accessed during nearest-neighbor or stencil computations, which are common in iterative methods for solving discretized partial differential equations.

Shadow regions, also referred to as *overlap areas* [12], may be specified for `BLOCK` or `GEN_BLOCK` distributions by means of the `SHADOW` directive by providing offsets with respect to the lower and upper bound of distributed dimensions according to the following syntax.

**Syntax: Shadow Directive**

| | | |
|---|---|---|
| *shadow-directive* | **is** | `SHADOW` *shadow-target shadow-attr-stuff* |
| *shadow-target* | **is** | *object-name* |
| | **or** | *component-name* |
| | **or** | *structure-component* |
| *shadow-attr-stuff* | **is** | ( *shadow-spec-list* ) |
| *shadow-spec* | **is** | *width* |
| | **or** | *low-width* : *high-width* |
| | **or** | * |
| *width* | **is** | *int-expr* |
| *low-width* | **is** | *int-expr* |
| *high-width* | **is** | *int-expr* |

Constraint: Any *int-expr* appearing in a *shadow-spec* of a *specification-directive* must be a *specification expression* with value greater than or equal to zero.

Constraint: The number of shadow-specs must match the rank of the *shadow-target*.

Constraint: A shadow-spec must be "*" if and only if the dist-format of the corresponding array dimension is "*".

A *shadow-spec* of *width* is equivalent to *width : width*.

The information provided by shadow areas simplifies the organization of communication as well as the local addressing significantly. Based on shadow specifications, the compiler can extend the local array section on each processor in order to hold data moved in from neighboring processors.

### 3.5.2 Halo Specifications

A halo region is an irregular, non-contiguous area describing the non-local accesses to a distributed array. Whereas shadow regions are usually specified in the context of `BLOCK` or `GEN_BLOCK` distributions, halo regions may be specified for *any* distribution. A halo region for a distributed array is specified by means of the *halo-directive* with the following syntax:

**Syntax:** The HALO Directive

| | | |
|---|---|---|
| *halo-directive* | **is** | `HALO` *halo-target halo-attr-stuff* |
| *halo-target* | **is** | *object-name* |
| | **or** | *component-name* |
| | **or** | *structure-component* |
| *halo-attr-stuff* | **is** | ( *halo-spec-list* ) `ON` *halo-proc-ref* |
| *halo-spec* | **is** | *int-array-expr* |
| | **or** | * |
| *halo-proc-ref* | **is** | *processors-name* ( *halo-dummy-list* ) |
| *halo-dummy* | **is** | *scalar-int-variable* |

Constraint: The number of *halo-spec*s must match the rank of *halo-target*.

Constraint: A *halo-spec* must be "*" if and only if the *dist-format* of the corresponding array dimension is "*".

Constraint: The *halo-proc-ref* must refer to the same processor array as the corresponding *halo-target* is distributed to.

Constraint: A *halo-spec* must be a rank one integer array expression where all elements have a value greater than or equal to the lower-bound and less-than or equal to the upper-bound of the corresponding dimension of the *halo-target*.

Constraint: Any *int-array-expr* appearing in a *halo-spec* of a *specification-directive* must be a *specification expression*.

Constraint: The number of *halo-dummy*s must match the rank of *processors-name*.

A *halo-spec* is a one-dimensional integer array expression (usually a function reference) that contains exactly one *halo-dummy variable*[4] which ranges from 1 to the number of processors in the corresponding processor array dimension. By evaluating the *halo-spec* for each processor, the set of non-local indices of the corresponding array that may be accessed during execution of the program is obtained. This information is utilized to organize the allocation of temporary storage for non-local data and to compute the associated communication schedules.

Halo areas may also be used for dynamically distributed arrays. In this case the `HALO` attribute has to be specified with the `REDISTRIBUTE` directive. This feature allows to change the halo of an array whenever its distribution is changed. Example 3.8 shows the use of a halo for a dynamically distributed array. In this example we assume that both the mapping array and the halo array are generated by a mesh partitioner and read in at runtime.

**Example 3.8: Halos for Dynamically Distributed Arrays**

```
      REAL, DIMENSION(N,M) :: A

      TYPE HALO_TYPE
         INTEGER, POINTER :: INDEX(:)
      END TYPE HALO_TYPE

      TYPE (HALO_TYPE), ALLOCATABLE :: SLHALO(:)

!HPF$ PROCESSORS :: R(NUMBER_OF_PROCESSORS())
!HPF$ DYNAMIC    :: A

      ALLOCATE(SLHALO(NUMBER_OF_PROCESSORS))

      READ(*,*) MAP                     ! mapping array

      DO I = 1, NUMBER_OF_PROCESSORS()
         READ(*,*) N                    ! number of non-local elements on R(I)
         ALLOCATE(SLHALO(I)%INDEX(N))
         READ(*,*) SLHALO(I)%INDEX(N) ! indices of non-local elements on R(I)
      END DO

!HPF+ REDISTRIBUTE(INDIRECT(MAP),*), HALO(SLHALO(I)%INDEX,*) ON R(I) :: A
```

*This example shows a runtime-defined indirect distribution with a halo. Both the mapping array* `MAP` *and the halo array are read prior to the distribution of* `A`*. The data structure* `SLHALO`*, an array of arrays defined by means of derived types, is used to store the halo information.* ■

[4]For *halo-dummy* variables the same conventions hold as for *alignment-dummy* variables.

## 3.6 Redistribution

The distribution of an object that has the `DYNAMIC` attribute may be changed at run time. Such objects are said to be *dynamically distributed*, as opposed to *statically distributed*. Whereas statically distributed objects have a fixed distribution within their scope of declaration, dynamically distributed objects may change the distribution within their scope of declaration.

### 3.6.1 Redistribution

An object with the `DYNAMIC` may be dynamically *redistributed* in the following ways:

- It may be redistributed by means of the executable directive `REDISTRIBUTE`.
- It may be redistributed as the result of a procedure call.

Any dynamically distributed object that is not explicitly aligned to another object may be redistributed. All objects that are ultimately aligned to an object that is being redistributed are also redistributed in order to maintain the alignment relationships.

Furthermore, an object with the `DYNAMIC` attribute that is passed as an argument to a procedure may have a different distribution after the procedure call if the corresponding actual argument also has the `DYNAMIC` attribute and has been redistributed within the procedure (see Section 7).

HPF+ extends the syntax of the `REDISTRIBUTE` directive such that it is possible to specify the `SHADOW` or `HALO` attribute for the arrays being redistributed. The syntax of the `REDISTRIBUTE` directive is specified in the Appendix. Specifying a halo within a `REDISTRIBUTE` directive is shown in Example 3.8 on the previous page.

### 3.6.2 The `RANGE` Attribute

In order to supply the compiler with information on the set of different distributions a dynamically distributed object (or a dummy argument with the `INHERIT` attribute) may be associated with, the `RANGE` attribute is provided.

The `RANGE` attribute may consist of an arbitrary number of distribution specifications. All these distributions are referred to as *range of permissible distributions*. Any distribution associated with an array must be compatible with at least one of the distributions specified in the range of permissible distributions. It is illegal to associate an object that has the `RANGE` attribute with a distribution not contained in the range of permissible distributions.

If the `RANGE` attribute is absent, a dynamically distributed object may be associated with *any* permitted distribution. The `RANGE` attribute may be specified according to the rules specified below. The compatibility of a *range-dist-format* and an associated *dist-format* is defined as shown in Table 2. The keyword `ALL` can be used in any place where a distribution format may occur with the meaning that it allows any legal substitution. Similarly, an argument of a distribution format may be omitted.

For example, `RANGE X(BLOCK())` indicates that the actual arrays associated with `X` have a `BLOCK(M)` distribution with any permitted value of `M`. Example 3.9 illustrates the use of the `RANGE` attribute.

The `RANGE` attribute may be also specified for dummy arguments with an *inherited distribution* as described in Section 7.

| *range-dist-format* | *dist-format* |
|---|---|
| BLOCK | BLOCK |
| BLOCK() | BLOCK, BLOCK(N), for any permissible value of N |
| BLOCK(M) | BLOCK(N), and N == M |
| GEN_BLOCK | GEN_BLOCK(B), and B is any permissible array |
| GEN_BLOCK(A) | GEN_BLOCK(B), and ALL(A==B) == .TRUE. |
| CYCLIC | CYCLIC |
| CYCLIC() | CYCLIC, CYCLIC(N), for any permissible value of N |
| CYCLIC(M) | CYCLIC(N), and N == M |
| INDIRECT | INDIRECT(B), and B is any permissible array |
| INDIRECT(A) | INDIRECT(B), and ALL(A==B) == .TRUE. |
| * | * |
| ALL | all permissible distribution formats |

Table 2: Compatibility of *range distribution formats*

The syntax of the RANGE directive is defined as follows:

**Syntax:** The RANGE Directive

| | | |
|---|---|---|
| *range-directive* | **is** | RANGE *object-name range-attr-stuff* |
| *range-attr-stuff* | **is** | *range-distribution-list* |
| *range-distribution* | **is** | ( *range-dist-format-list* ) [ *dist-onto-clause* ] |
| | **or** | ( *dist-onto-clause* ) |
| | **or** | ( ALIGN *align-attribute-stuff* ) |
| *range-dist-format* | **is** | BLOCK [ ( [ *int-expr* ] ) ] |
| | **or** | CYCLIC [ ( [ *int-expr* ] ) ] |
| | **or** | GEN_BLOCK [ ( *int-array-expr* ) ] |
| | **or** | MULTI_BLOCK [ ( *int-array-expr*, *int-array-expr* ) ] |
| | **or** | INDIRECT [ ( *int-array-expr* ) ] |
| | **or** | * |
| | **or** | ALL |

Constraint: An object that has the RANGE attribute must have the DYNAMIC attribute or the INHERIT attribute.

Constraint: The number of *range dist-formats* must be equal to the rank of *object*.

Constraint: Any *int-expr* appearing in a *range-dist-format* attribute must be a specification expression.

Constraint: Any *int-array-expr* appearing in a *range-dist-format* must be a specification expression.

**Example 3.9: Range Attribute**

```
!HPF$ PROCESSORS R(8)

      REAL, DIMENSION(N,N) :: A
      REAL, DIMENSION(N,N) :: B
      REAL, DIMENSION(N)   :: C)

!HPF+ DYNAMIC, RANGE (*,*), (ALIGN WITH B)   :: A
!HPF+ DYNAMIC, RANGE (BLOCK,*), (*,BLOCK)    :: B
!HPF+ DYNAMIC, RANGE (ONTO R(1:4))           :: C
```

*The range directives of this example constitute an assertion to the compiler that* `A` *and* `B` *are always identically aligned.* `B` *may be partitioned into blocks of columns or blocks of rows, respectively.* `C` *may be distributed arbitrarily, provided that all distributions are restricted to the first 4 processors of* `R`. ■

## 3.7 Allocatable Arrays and Pointers

Allocatable arrays are handled in HPF+ in exactly the same way as in HPF. A variable with the `ALLOCATABLE` attribute may appear as an *alignee* in an `ALIGN` directive or as a *distributee* in a `DISTRIBUTE` directive.

In contrast to HPF, where objects that have the `POINTER` attribute must not be distributed, in HPF+ the distribution of pointer objects may be specified in the same way as for allocatable arrays.

### 3.7.1 Dynamic Memory Allocation

Distribution directives specified for an object that has the `ALLOCATABLE` or the `POINTER` attribute do not take effect immediately when the declaration is evaluated; they take effect each time such an object is allocated by means of an `ALLOCATE` statement. The values of all specification expressions in such directives are determined once on entry to the scoping unit and are retained until the time of allocation or pointer assignment.

If an allocatable array is an alignee, then at the time of allocation the align-target must already be allocated.

As in Fortran, the allocation of an actual array that has the `POINTER` attribute may be performed within a procedure, if and only if the corresponding dummy argument also has the `POINTER` attribute. If in addition both the actual and the dummy argument have the `DYNAMIC` attribute, then the distribution of the actual array after the procedure call will be according to the distribution of the dummy argument. Changing the distribution of objects as the result of procedure calls is described in more detail in Section 7.

### 3.7.2 Pointer Assignment

A statically distributed pointer may be associated with a target by means of a pointer assignment statement if and only if the target is statically distributed, and has an identical distribution.

A dynamically distributed pointer may be associated with a target by means of a pointer assignment statement if and only if the target has the `DYNAMIC` attribute. In such a case the pointer takes over the distribution of the target.

A dynamically distributed pointer may be redistributed by means of `REDISTRIBUTE` if and only if it is currently associated with a whole array, not an array section. When an object is redistributed then any pointer that my be associated with that object also acquires that changed distribution.

## 3.8 Distribution of Derived Type Components

A derived type that has a component for which a `DISTRIBUTE`, `ALIGN`, or `DYNAMIC` is specified is called a *distributed derived type*. Any such directive may be specified for a component of a derived type if and only if it is of an intrinsic type or if it is a structure and none of its components (at any level of component selection) is of a distributed derived type. Similarly, a variable of a derived type can be explicitly distributed iff none of the components of the derived type are distributed.

Example 3.10 shows the declaration of a multi-grid data structure which can be used to implement an arbitrary number of grids of possibly different (run time defined) sizes.

**Example 3.10: Distribution of Derived Type Components**

```
      TYPE SUBGRID
          INTEGER XSIZE, YSIZE
          REAL, DIMENSION(:,:), POINTER :: GRID
!HPF$           DISTRIBUTE(BLOCK,BLOCK) :: GRID
          ...
      END TYPE SUBGRID

      TYPE(SUBGRID), DIMENSION(:), ALLOCATABLE :: GRID_STRUCTURE
```

*This example shows the declaration of a variable* `GRID_STRUCTURE` *which is applicable to multigrid codes which require that the number and sizes of grids are determined at run time. Since the derived type* `SUBGRID` *has a distributed component, it would be illegal to specify a distribution for the variable* `GRID_STRUCTURE`*.* ■

## 3.9 Discussion

The data distribution mechanism of HPF-2 is the same as the basic data distribution mechanism of HPF+. In addition, HPF+ provides distribution to processor subsets, general block distributions, multi-block distributions, indirect distributions and dynamic distributions. These extensions, except for multi-block distributions, have also been adopted by the Approved Extensions of HPF-2. Note, however, that the concept of processor views of HPF+ provides more flexibility when specifying distributions to processor subsets.

Moreover, HPF+ extends the `DISTRIBUTE` directive such that replication occurs if the number of distributed dimensions of an array is less than the rank of the processor array to which the distribution refers. This is similar to the distribution mechanism of Vienna Fortran.

The alignment mechanisms have been simplified without loss of required functionality. Unlike HPF, in HPF+ the distribution of an alignee can always be represented by means of a direct distribution.

Distribution of components of derived types has been defined initially in Vienna Fortran 90 based on the concept of *distributed derived types*. The HPF+ mechanisms for distribution of components of a derived type are also included in the Approved Extensions of HPF-2.

# 4 Independent Directive

The base language of HPF+, Fortran 95, provides several constructs such as array assignments and `FORALL` to specify data parallel operations on arrays. Although these constructs do not define the order in which individual array elements are assigned, they exhibit a *fetch-before-store* semantics. In order to perform important optimizations, the compiler must ensure that the semantics of an array assignment is preserved without enforcing the evaluation of the right hand side prior to defining the left hand side.

Similarly, many parallelizing loop transformations require that the iterations may be performed in any order. However, in many cases, and particularly if arrays are accessed indirectly, as in most irregular codes, this information cannot be derived by the compiler. Thus the user may provide this information by means of the `INDEPENDENT` directive which may be specified in HPF+ not only for `DO` loops and `FORALL` statements, but also for array assignment statements.[5]

In order to parallelize loops, forall statements and array assignments efficiently, the iteration space or index space of these constructs has to be distributed to the available processors with the goal of minimizing the communication while attempting to balance the load. This task, referred to as work distribution, decisively influences a program's performance. Usually work distribution is determined by the compiler according to the *owner-computes* rule. If data distributions or access patterns are irregular or dependent on runtime data, work distribution based on compile-time information may result in a poor data locality or bad balance of work load. Therefore, HPF+ enables the user to specify the work distribution. The features provided for an efficient parallelization of irregular loops, forall statements and array assignments are described next.

## 4.1 Independent `DO` Loops

Whenever a loop contains indirect array accesses, the compiler will usually not be able to determine whether the iterations of the loop may be executed in parallel. Since such loops are common in irregular problems, and may contain the bulk of the computation, the user must assert the independence of its iterations.

The `INDEPENDENT` directive allows the programmer to assert that no data object written by one iteration is read or written by another, and thus the iterations of a `DO` loop may be executed in any order.

The `NEW` clause may be used to introduce *private variables*. Private variables are conceptually local in each iteration, and therefore do not cause loop-carried dependences. All variables specified in a `NEW` clause are conceptually undefined immediately before and after execution of every loop iteration. Note that if a procedure is called from an `INDEPENDENT` loop, then any local variables are considered to be distinct on each call unless they have the `SAVE` attribute.

The `REDUCTION` clause and *reduction statements* support the specification of reduction operations without introducing dependences.

The distribution of loop iterations to processors may be specified either directly or according to the distribution of arrays by appending an `ON` clause or `ON HOME` clause to the `INDEPENDENT` directive. The syntax of the HPF+ `INDEPENDENT` directive, not showing the features for communication schedule reuse, is as follows:

[5] In HPF the `INDEPENDENT` directive cannot be specified for array assignments.

| | | |
|---|---|---|
| *independent-directive* | **is** | `INDEPENDENT` [ *independent-clause-list* ] |
| *independent-clause* | **is** | *new-clause* |
| | **or** | *reduction-clause* |
| | **or** | *on-clause* |
| | **or** | ... |
| *new-clause* | **is** | `NEW` ( *variable-list* ) |
| *reduction-clause* | **is** | `REDUCTION` ( *reduction-variable-list* ) |
| *reduction-variable* | **is** | *array-variable-name* |
| | **or** | *scalar-variable-name* |
| | **or** | *structure-component* |
| *on-clause* | **is** | `ON HOME` ( *variable* ) |
| | **or** | `ON` ( *processors-section* ) |

Constraint: The first non-comment line following an *independent-directive* must be a *do-stmt*, *forall-stmt*, a *forall-construct*, or an array assignment statement.

Constraint: If either the *new-clause* or the *reduction-clause* is present, then the *independent-directive* must apply to a *do-stmt*.

Constraint: A *variable* named in the *new-clause* and any component or element thereof must not:

- Be a dummy argument.
- Have the `SAVE` or `TARGET` attribute.
- Be storage associated with another object.
- Be use associated or host associated.
- Be accessed in another program unit via host association.

Constraint: A *reduction-variable* must be of intrinsic type.

Constraint: A *variable* must not both appear in the *new-clause* and the *reduction* clause of the same *independent* directive.

Constraint: An *variable* referenced in the *on-clause* of an `INDEPENDENT` directive of a *do-stmt* must be an array variable that references in exactly one subscript the loop iteration variable of the *do-stmt*.

Constraint: An *variable* referenced in the *on-clause* of a *forall-stmt* or *forall-construct* must be an array element or section that references all *forall-indices*, each in a different array dimension.

Constraint: An *variable* referenced in the *on-clause* of a array assignment statement must be an array section that is conformable with the left-hand-side array section of the assignment statement.

Extensions of the `INDEPENDENT` directive for communication schedule reuse, and for locality assertions are specified in Section 5 and 6, respectively.

### 4.1.1 Reduction Operations

The `REDUCTION` clause and *reduction statements* support the specification of reduction operations within `INDEPENDENT DO` loops. If a variable is subject to a reduction operation it must be specified within the `REDUCTION` clause. Any variable whose name occurs in a reduction clause is said to be *protected* while the immediately following loop is *active*. A protected reduction variable may be referenced only in a *reduction statement* which must satisfy certain syntactic and semantic constraints as described below. In particular, it may not occur in any directive including the variable list in a `NEW` clause. This includes any `NEW` clause in the same directive. Note that, within the `REDUCTION` clause a *reduction variable* must not be a *subobject*, even though an array element or array section may occur in a reduction statement.

A reduction statement is an assignment statement that occurs in the range of an independent `DO` loop for which the name of its reduction variable occurs in a reduction clause. Reduction statements must be of the following form:

**Syntax: Reduction Statements**

| | | |
|---|---|---|
| *reduction-stmt* | **is** | *reduction-var-ref* `=` *expr reduction-op reduction-var-ref* |
| | **or** | *reduction-var-ref* `=` *reduction-var-ref reduction-op expr* |
| | **or** | *reduction-var-ref* `=` *reduction-function*`(`*reduction-var-ref*`,` *expr*`)` |
| | **or** | *reduction-var-ref* `=` *reduction-function*`(`*expr*`,` *reduction-var-ref*`)` |
| *reduction-var-ref* | **is** | *variable* |
| *reduction-op* | **is** | *intrinsic-op* |
| *reduction-function* | **is** | `MAX` |
| | **or** | `MIN` |
| | **or** | `IAND` |
| | **or** | `IOR` |
| | **or** | `IAND` |

Constraint: A reduction variable must not be a substring.

Constraint: The *reduction-var-refs* in a given *reduction-stmt* must be lexically identical.

Constraint: *expr* must not contain a reference to *reduction-var*.

Constraint: A *reduction-op* must be an associative and commutative operator.

Reduction variables may only be referenced within reduction statements as outlined above. In particular, a reference to a reduction variable must not occur in the *expr*-part of the reduction statement or in a procedure that is called from the range of the corresponding independent loop.

On exit from an independent loop with a reduction clause, the value of the reduction variable is processor dependent (i.e. it may be a processor dependent approximation of the value that would have been computed by sequential execution).

In nested loops, a reduction-clause must appear on the outermost independent loop that has no *new-clause* for the reduction variable.

For more details on reduction operations we refer the reader to the HPF language specification. Specifying reduction operations within `INDEPENDENT` loops is shown in Example 4.1.

**Example 4.1: Reduction Directive**

```
!HPF$ INDEPENDENT, REDUCTION(X)
DO I = 1, N
   X(IDX(I)) = X(IDX(I)) + F(I)
END DO
```

■

### 4.1.2 Work Distribution

In order to efficiently parallelize loops that access distributed data objects the iteration space has to be distributed to the available processors with the goal of minimizing the communication (i.e. non-local data accesses) while attempting to balance the load. Usually the distribution of loop iterations is determined by the compiler by analyzing the accesses to the distributed arrays. However, if data access patterns are dependent on run time data, the distribution of loop iterations by the compiler may imply a high run time overhead.

HPF+ addresses this issue by enabling the user to specify how the iterations of an independent loop should be distributed. By means of an *on-clause* the iterations may be assigned to the processors either directly or according to the distribution of data objects. This mechanism, also referred to as *work distribution*, is at the iteration level, i.e., full iterations may be assigned to a single processor.

By means of an *on-clause* the user may provide additional information to the compiler on how to assign the iterations of an independent `DO` loop to the processors. Two different forms of the *on-clause* are provided. An on-clause of the form `ON HOME(A(i))` specifies that iteration $i$ should be executed on the processor that owns the array element `A(i)`. If `A(i)` is replicated across several processors, a system-dependent processor will be chosen. It is also possible to name the processor directly, by using an on-clause of the form `ON(R(i))` where `R` must designate a processor array.

**Example 4.2: `ON` and `ON HOME` Clause**

```
!HPF$ PROCESSORS R(10)
!HPF$ DISTRIBUTE C(CYCLIC)
!HPF$ ALIGN b WITH c

!HPF+ INDEPENDENT, ON HOME(B(IDX(I)))
      DO I = 1, N
         A(IDX(I)) = B(IDX(I)) + C(IDX(I))
      END DO

!HPF+ INDEPENDENT, ON R(MOD(I,10))
      DO I = 1, N
        A(I) = C(IDX(I))
      END DO
```

*The on-clause of the first independent directive specifies that the $I^{th}$ loop iteration is to be executed on the processor that owns array element* `B(IDX(I))`*. The second on-clause specifies that the $I^{th}$ iteration should be executed on processor* `R(MOD(I,10))` *which results in a round-robin (i.e. cyclic) distribution of loop iterations.* ■

In case of nested independent loops an *on-clause* may be specified for each independent loop. In this case, the processors designated by the *on-clause* of an inner independent loop must be a subset of the processors specified by on-clauses of enclosing independent loops as shown in Example 4.3.

**Example 4.3: Nested `ON`-Clauses**

```
!HPF+ INDEPENDENT, NEW(J), ON HOME(A(I,:))
      DO I = 2, N, 2
!HPF+    INDEPENDENT, ON HOME(A(I,J))
         DO J = 1, N
            A(I,J) = A(I-1,J) + A(I,J)
         END DO
      END DO
```

*This example shows the use of on-clauses for nested independent loops. The processors specified by the on-clause of the inner loop are always a subset of the processors specified within the outer independent directive. Note that the iteration variable of the inner loop has to be included in a* `NEW` *clause of the outer independent directive.* ■

## 4.2 Independent `FORALL` Statement and Construct

An `INDEPENDENT` directive in the context of a `FORALL` statement or construct asserts that no combination of the `FORALL` indices assigns to the same array element repeatedly, or to an array element read by any other combination. In order to partition the index space of `FORALL` statements or constructs, the `ON` or `ON HOME` clause may be used, either in connection with an `INDEPENDENT` directive or stand-alone, to specify the processors responsible for assigning to individual array elements.

## 4.3 Independent Array Assignments

For array assignment statements, including `WHERE` statements, the `INDEPENDENT` directive may be used to assert that neither are array elements defined several times, nor is any array element both defined and read. The `ON HOME` clause may be used to specify that the work distribution should be derived depending on the ownership of a particular array or array section accessed. The use of the `INDEPENDENT` directive for array assignment statements is shown in Example 4.4.

## 4.4 Discussion

The *independent loop* concept proposed for HPF+ is based on the HPF-2 features outlined above (i.e. `INDEPENDENT` directive with optional `NEW` clause and `REDUCTION` clause), together with additional mechanisms for the specification of work distribution.

These additional language features have been mainly motivated by Vienna Fortran which provides support for work distribution and reduction operations. In Vienna Fortran the loop iteration space may be distributed according to the owner(s) of distributed (sub)objects, to particular processors, or using a `BLOCK` or `CYCLIC` distribution [8]. For specifying reduction operations, a new executable statement is provided which in addition to intrinsic reduction operations also allows user-defined reductions.

**Example 4.4: Independent Array Assignments**

```
!HPF$ REAL, DIMENSION(N1) :: A
      REAL, DIMENSION(N2) :: B, C

!HPF$ PROCESSORS R(NUMBER_OF_PROCESSORS)
!HPF$ DISTRIBUTE(GEN_BLOCK(BS))  :: A
!HPF$ DISTRIBUTE(BLOCK)          :: B, C

!HPF$ INDEPENDENT, ON HOME(B(N:M))
      A(L1:U1) = A(L2:U2) + B(N:M) + C(N:M)
```

*This example shows an array assignment statement with an* `INDEPENDENT` *directive and an* `ON HOME` *clause. The* `INDEPENDENT` *directive in this example asserts that the intersection of the array sections* `A(L1:U1)` *and* `A(L2:U2)` *is empty. The* `ON HOME` *clause specifies that the work distribution should be derived according to the distribution of the array section* `B(N:M)`*. For example, the assignment to* `A(L1)` *will be performed on the processor owning* `B(N)`*.* ∎

The HPF-2 Approved Extensions, introduce the `ON HOME` and `ON` directive which can be used by the programmer to recommend where operations should be executed. A free-standing on-directive, which applies to the immediately following statement, as well as a block-structured variant are provided. The `ON HOME` and `ON` directive are more general than the on-clause of Vienna Fortran. In HPF-2 they have been primarily designed in order to provide support for the specification of task parallelism.

The syntax and semantics of HPF+ independent loops is in correspondence with the language provided by HPF-2 and its Approved Extensions with the exception that an additional *work-distribution-clause* may be specified. The on-clause of HPF+ is similar to the free-standing on-directive of the HPF-2 Approved Extensions. Furthermore in HPF+, new-clause, reduction-clause, on-clause, and distribute-clause may be specified in any order.

# 5 Support for Communication Schedule Reuse

Schedules are internal communication objects that have to be constructed by the runtime system when a parallel loop with irregular array accesses is entered. This is done in a preprocessing phase called the "inspector". Informally, a schedule may be described as the set of non-local elements of a distributed array that are accessed during execution of a parallel loop.

For a parallel loop with indirect data accesses, an *inspector* analyzes the data access pattern at runtime. From the access pattern, the array distribution, and the work distribution of the loop, a communication schedule can be derived. This schedule is then used in the *executor* to actually perform the required communication for the loop by gathering all nonlocal data to be read in the loop at the beginning, and scattering all nonlocal data that were written in the loop, at the end.

Building a schedule may be very expensive, in some cases dominating the execution time of the program. Often, this cost can be amortized across subsequent executions of a parallel loop (for example, if the parallel loop is enclosed in a sequential time-step loop) or across different loops with similar access patterns. The recognition of invariant communication schedules is crucial for obtaining high object code performance. In some cases, a compiler can recognize this invariance automatically. However, often such loop structures are hidden in a hierarchy of procedure calls so that even sophisticated interprocedural analysis may be unable to achieve this goal. Runtime analysis techniques are similarly restricted. Thus, *language* support is required for this problem.

The main motivation for the language extensions described in the following is to allow the user explicit control over the generation of communication schedules, thus avoiding the redundant computation of communication schedules by an inspector.

## 5.1 The `REUSE` Clause

The redundancy of a schedule computation in the context of an `INDEPENDENT` loop can be expressed by specifying the `REUSE` clause. This constitutes an assertion to the compiler that the communication schedules of all arrays accessed are invariant for *all* executions of the loop and thus have to be computed only once, when the loop is executed the first time. The invariance of the communication schedules of a loop implies

- that the loop iteration space is invariant,
- that all distributed arrays referenced within the loop have the same distribution,
- and that all indirection arrays used in subscripts of distributed arrays have the same contents whenever the loop is executed.

Note that this also applies to arrays referenced in the `ON HOME` clause, which must be present if the `REUSE` clause is specified. Optionally, a reuse-clause may be followed by a list of distributed arrays. In this case the reuse of communication schedules is restricted to the arrays specified in the list.

In order to handle conditional schedule reuse, a condition in the form of a scalar logical expression may be associated with the `REUSE` clause. If a `REUSE` clause contains a condition, then the schedules are reused if and only if the condition evaluates to *true*. This feature can be used to permit the reuse of communication schedules within certain different phases of a program, while forcing recomputation of schedules between these phases, as for example in adaptive problems. The syntax and examples of the `REUSE` clause are presented on the next page.

The syntax for the REUSE clause is as follows:

| | | |
|---|---|---|
| *reuse-clause* | **is** | REUSE [ ( *array-list* [ [ COND = ] *scalar-logical-expr* ] ) ] |

**Example 5.1: The REUSE Clause**

```
      DO T = 1, MAX_TIME
         CALL KFORCE(F,X,IX,...)
         ...
      END DO
      ...
      SUBROUTINE KFORCE(F,X,IX,...)
         ...
!HPF$ INDEPENDENT, REUSE
         DO I = 1, NUMEL
            ... = x(I,IX(1,I)) ....
         END DO
         ...
      END SUBROUTINE KFORCE
```

*The* REUSE *clause in this example (which is an excerpt of the HPF+ PAM-CRASH benchmark kernel) has the effect that the communication schedules of all distributed arrays accessed within the loop in procedure* KFORCE *is executed only the first time the* INDEPENDENT *loop is executed.* ■

## 5.2 Schedule Variables

Schedule variables are introduced in order to provide the user with explicit control over construction, application, and reuse of schedules.

### 5.2.1 Declaration of Schedule Variables

Schedule variables are declared by means of the SCHEDULE directive which must occur in the specification part of a program unit. At the time of declaration a schedule variable is set to undefined[6].

**Syntax:** The SCHEDULE Directive

| | | |
|---|---|---|
| *schedule-directive* | **is** | SCHEDULE [ [, *schedule-attr-spec-list* ] :: ] *schedule-decl-list* |
| *schedule-decl* | **is** | *schedule* [ ( *schedule-shape-spec-list* ) ] |
| *schedule-attr-spec* | **is** | DIMENSION ( *schedule-shape-spec-list* ) |
| | **or** | SAVE |
| *schedule* | **is** | *object-name* |
| *schedule-shape-spec* | **is** | *explicit-shape-spec* |

[6] This may not be the case if a schedule variable has the SAVE attribute.

**Example 5.2: The `REUSE` Clause**

```
         DO T = 1, MAX_TIME_STEPS
!HPF$       INDEPENDENT, ON HOME(B(Y(I))), REUSE(MODULO(T,10) /= 1)
            DO I = 1, N
               A(X(I)) = ... B(Y(I)) ...
            END DO
            ...
         END DO
```

*The optional condition associated with the* `REUSE` *clause implies that the communication schedule required for non-local accesses to* `A` *and* `B` *will be (re)computed in iteration* $1, 11, 21, ...,$ *of the outer loop. In all other iterations the communication schedules will be reused. Note that the* `REUSE` *clause could also be used if the inner loop were located in a procedure called from with the time-step loop.* ■

Note that schedule variables are not first class, and thus must not be arguments of procedures. However, schedule variables that are declared in a module are accessible from procedures by means of use-association.

### 5.2.2 Assignment and Use of Schedule Variables

Schedule variables may be associated with arrays in the context of `INDEPENDENT` loops by means of the `GATHER` and/or `SCATTER` clause.

**Syntax:** The `GATHER` and `SCATTER` Clause

| | | |
|---|---|---|
| *gather-clause* | **is** | `GATHER (` *schedule-def-use-list* `)` |
| *scatter-clause* | **is** | `SCATTER (` *schedule-def-use-list* `)` |
| *schedule-def-use* | **is** | *object-list* `::` *schedule-variable* |

At the time an `INDEPENDENT` loop $L$ is executed a gather-clause of the form `GATHER(A::S)` has, depending on whether `S` is *undefined* or *defined*, the following effect.

- *Def-Use Semantics*
  If the schedule variable `S` is *undefined*, then the schedule determined for `A` (which is usually determined by an inspector) is assigned to `S` and on each processor all non-local elements of `A` *read* in $L$ are gathered according to `S`.
- *Use Semantics*
  If the schedule variable is already *defined*, on each processor the non-local elements of `A` *read* in $L$ are gathered as prescribed by `S`.

Similarly, a `SCATTER` clause is utilized to define and/or use schedules for arrays that are *written* in a loop. Schedule reuse for a parallel loop that is executed within an outer time-step loop is shown in Example 5.3.

**Example 5.3: Schedule Reuse**

```
!HPF$ ALIGN B WITH C

!HPF$ SCHEDULE :: S     ! declaration of schedule variable S
...

DO K = MAX_TIME
   ...
   !HPF+ INDEPENDENT, ON HOME(A(X(I))), GATHER(B,C::S)
   DO I = 1, N
      A(X(I)) = B(X(I)) + C(X(I))
   END DO
   ...
END DO
```

*At the time execution reaches the inner loop first, the schedule variable is undefined. The* `GATHER` *clause results in the execution of the inspector for the loop and the resulting schedule for* `B` *and* `C` *is assigned to* `S`. *On subsequent iterations of the outer loop the schedule bound to* `S` *is reused. Note that for array* `A` *no communication will be required since the* `ON HOME` *clause implies that only local elements of* `A` *are written on each processor.* ■

If a schedule variable declared in a procedure has the `SAVE` attribute its value is retained. Thus, the communication schedules used inside procedures may be reused between subsequent incarnations. This is shown in Example 5.5, where a procedure that contains parallel loops with indirect array accesses is called from within a time-step loop located in a different procedure.

### 5.2.3 Conditional Schedule Reuse with Schedule Variables

By means of the `RESET` directive the user may explicitly *undefine* a schedule variable. Thus, the reuse of schedules may be controlled by the user depending on runtime-conditions. This is in particular important for applications that exhibit different communication patterns in different program phases. The `RESET` directive is an executable directive which may occur anywhere in the execution part of a program unit. In Example 5.4 conditional reuse of schedules is shown.

**Syntax:** The `RESET` Directive

*reset-directive* **is** `RESET` [::] *schedule-variable-list*

**Example 5.4: Conditional Schedule Reuse with the RESET Directive**

```
!HPF+ SCHEDULE :: S

      DO K = MAX_TIME
!HPF$    INDEPENDENT, ON HOME(A(X(I))), GATHER(B,C::S)
         DO I = 1, N
            A(X(I)) = B(X(I)) + C(X(I))
         END DO
         IF (RECONFIGURE(...)) THEN
            CALL RECOMPUTE(X)
!HPF$       RESET :: S
         END IF
         ...
      END DO
```

*At the time execution reaches the inner loop first, the schedule variable is undefined. The* `GATHER` *clause results in the execution of the inspector for the loop and the resulting schedule is assigned to* `S`*. On subsequent iterations of the outer loop the schedule bound to* `S` *is reused as long as the logical function* `RECONFIGURE` *does not evaluate to* `.TRUE.`*. If* `RECONFIGURE` *evaluates to* `.TRUE.`*, the* `RESET` *directive sets the schedule variable* `S` *to* undefined *and the inspector is executed again in the next iteration of the outer loop.* ∎

**Example 5.5: Reusing Schedules in Procedures**

```
SUBROUTINE IRREG(X,Y,Z,IDX1,IDX2)
!HPF+ SCHEDULE, SAVE :: S1
      ...
!HPF$ INDEPENDENT, ON HOME(Y(IDX2(I))), SCATTER(X::S1)
      DO I = 1, MAX
         X(IDX1(I)) = Y(IDX2(I)) ...
      END DO
!HPF$ INDEPENDENT, ON HOME(Z(IDX2(I))), GATHER(X::S1)
      DO I = 1, MAX
         Z(IDX2(I)) = X(IDX1(I))  ...
      END DO
end subroutine irreg
```

*When the procedure* `IRREG` *is called first, the schedule variable* `S1` *is undefined. Thus, it gets associated with the schedule required for scattering non-local elements of* `X` *which is determined by executing an inspector for the first loop. Due to* `ON HOME`*-clause no communication is needed for array* `Y` *in the first loop. Schedule* `S1` *is reused in the second loop to gather non-local elements of* `X`*. On subsequent calls to the procedure the communication schedules for X in both loops will be reused since* `S` *has the* `SAVE` *attribute.* ∎

# 6 Locality Assertions

HPF+ provides the user with language features to assert that the parallelization of certain language constructs may be accomplished without the need to generate inter-processor communication. This includes the `PUREST` attribute and the `RESIDENT` attribute which are described next.

## 6.1 The `RESIDENT` Clause

The `RESIDENT` clause may be used within an `INDEPENDENT` directive in combination with the `ON HOME` clause to specify that the data accessed in the corresponding loop, forall, or array assignment is local to the executing processors, and thus no communication is required. If the resident-clause includes a list of arrays, then the locality assertion refers only to the specified arrays. This information, which the compiler may not be able to derive automatically, is utilized to avoid the generation of unnecessary communication. Note that the semantics of the `RESIDENT` clause, when used within the `INDEPENDENT` directive is different from the `RESIDENT` clause, directive, or construct provided in the context of task parallelism as described in Section 8.

**Example 6.1: The `RESIDENT` Clause**

```
!HPF$ INDEPENDENT, ON HOME(LB(I)), RESIDENT
          DO I=IA,IE
            PS(LB(I))=1.0
            PS(LCC(LB(I)+IS))=1.0
          END DO
```

*This example shows a loop from the FIRE HPF+ benchmark kernel. The* `RESIDENT` *clause asserts that all accesses to the distributed array* `PS` *are local if the work distribution of the loop is determined according to the specified* `ON HOME` *clause, i.e. loop iteration* `I` *is executed on the owner of element* `LB(I)`*. Thus, no communication is required.* ■

## 6.2 Purest Procedures

HPF-1 introduced the `PURE` attribute in order to enable the user to write procedures that are free of side-effects. Pure procedures are important in the context of data parallel constructs. In particular, a procedure must be pure if it is invoked from within a `FORALL` loop or construct. Pure procedures have been included in the new Fortran 95 Standard.

Pure procedures must obey a number of syntactic constraints which have to be checked by the compiler. Besides these constraints, a pure procedure is not allowed to perform any redistribution of global data or local data objects. As a consequence, dummy arguments may not appear in distribution directives that fix their location with respect to a processor array. Thus, dummy arguments of pure procedures may be explicitly aligned only with other dummy arguments, but must not have the `DISTRIBUTE`, `DYNAMIC`, or `INHERIT` attribute.

However, despite all these constraints, the invocation of pure procedures does not imply that no communication is required since they may, for example, read distributed global data objects. Thus, for a parallelizing compiler the `PURE` attribute is of limited use. Calling, for example, a procedure that accesses distributed global data objects from within an `INDEPENDENT` loop may result in a serialization of the loop because communication may be required to reference the global data.

For an efficient implementation of INDEPENDENT loops that access distributed objects it is crucial for the compiler to "know" that a procedure invocation does not cause communication. Thus, HPF+ extends the concept of pure procedures by providing *purest procedures*.

A *purest procedure* is a pure procedure with the additional constraint that its invocation does not require communication. More precisely, the invocation of a purest procedure must not result in communication on entry, during execution, and upon exit from the procedure. Whenever a purest procedure is called all arguments are assumed to be already local on the processors executing the call. This information can be exploited by the compiler for an efficient parallelization of INDEPENDENT loops that contain procedure calls.

The PUREST attribute may be specified by means of the PUREST directive in the specification part of a procedure according to the following syntax.

**Syntax: The PUREST Directive**

*purest-directive* **is** PUREST

Constraint: The PUREST directive may only be specified for pure procedures.

Constraint: Purest procedures must not contain any HPF directives except the *purest-directive*.

The interface of a purest procedure must be explicit in the scope of its use. Thus for external procedures an interface block must be provided and that interface must specify the PUREST attribute.

A purest procedure must not contain any HPF directives, except the PUREST directive. Any procedure called (directly or indirectly) from a purest procedure is considered to be purest by default.

Example 6.2 which is an excerpt of one of the HPF+ PAM-CRASH kernels shows the use of purest procedures in the context of independent loops.

**Example 6.2: Purest Procedures**

```
!HPF$ DISTRIBUTE (*,BLOCK) :: X, IX
      ...
!HPF$ INDEPENDENT, NEW(xn,force)
      DO IEL = 1, NUMEL2
         XN = X(:,IX(:,IEL))     ! assign to private array
         CALL MFORCE(FORCE, XN)  ! call purest procedure
         ...
      END DO
      ...
      PURE SUBROUTINE MFORCE(FORCE, X)
!HPF$ PUREST
        ...
      END SUBROUTINE MFORCE
```

*The procedure* MFORCE*, called from within an* INDEPENDENT *loop, is declared to be purest. Before the call to* MFORCE *the relevant elements of the distributed array* X *are gathered into the array* XN *which is declared as* NEW *and thus private to the processor executing the call. The subroutine* MFORCE *performs local computations only and returns the result in the private array* FORCE *which has also been declared as* NEW*.* ■

# 7 Procedures

The distribution of statically distributed arrays never changes as the result of a procedure call. Thus, upon return from a procedure all statically distributed actual arguments are distributed in exactly the same way as they were before the call.

Dynamically distributed actual arrays, however, may have a modified distribution after a procedure call if the corresponding dummy array has the `DYNAMIC` attribute and has been redistributed inside the procedure. Whereas this allows to define or modify the distribution of dynamically distributed arrays inside procedures, unrestricted use of this feature may seriously complicate the task of the compiler to determine which distributions may reach a particular array reference and thus requires sophisticated interprocedural analysis.

Whenever distributed arguments are passed to procedures the compiler may have to generate a redistribution in order to enforce the distribution of the corresponding actual arguments. Similarly, such an implicit redistribution may be required upon exit if it is necessary to restore the original distribution of the actual arguments. This is described in detail in Section 7.4.

## 7.1 Interfaces

It is recommended to provide an interface block for all external procedures used in an HPF+ program. In particular, if it might be necessary to perform a redistribution when a procedure is called, the procedure's interface must be explicit[7] to the caller.

The interface of a procedure must be explicit if one of the following conditions hold:

- Fortran requires one. For example, if a procedure has a dummy argument that is an assumed shape array, or that has the `POINTER` or `TARGET` attribute, or if the result is array valued or a pointer.
- The procedure has a dummy argument with an *inherited* distribution.
- The procedure has a dummy argument with an *explicit* distribution, which, in at least one incarnation of the procedure, might be different from the distribution of the corresponding actual argument.
- The procedure has a dummy argument with the `DYNAMIC` attribute.

Whenever a redistribution of arguments is required on entry to or upon exit from a procedure the compiler will arrange for redistributing the actual argument to the specified distribution and possibly distributing it back on return from the subprogram, if necessary.

It is recommended to specify for all dummy arguments the `INTENT` attribute. This information can be exploited by the compiler to avoid unnecessary data motion at procedure boundaries. For objects with an `INTENT(OUT)` attribute no data movement on entry to a procedure is necessary. The same is true for objects with the `INTENT(IN)` attribute upon exit from a procedure.

[7] A procedure has an explicit interface within the scope of the caller, if it is an internal procedure (host association), a module procedure (use association), an intrinsic procedure or an external procedure for which an interface block is provided or accessible.

**Example 7.1: Explicitly Distributed Dummies**

```
!HPF$ PROCESSORS :: R(NUMBER_OF_PROCESSORS)
      REAL, DIMENSION(N)  :: A,B
!HPF$ DISTRIBUTE (BLOCK)  ONTO R :: A
!HPF$ DISTRIBUTE (CYCLIC) ONTO R :: B
      ...
      CALL SUB(A,B,...)
      ...
CONTAINS
      ...
      SUBROUTINE SUB(X, Y,...)
      REAL, DIMENSION(:), INTENT(IN) :: X, Y
      !HPF$ DISTRIBUTE(BLOCK) ONTO R :: X, Y
      ...
```

*In this example, the dummy arrays* X *and* Y *are explicitly mapped. Since the distribution of the dummy array* X *matches the distribution of the actual array* A *no redistribution will be required when* sub *is called. The second dummy array* Y *has a different distribution than the corresponding actual array* B*. As a consequence, a redistribution has to be performed on entry to the procedure in order to enforce the distribution specified for* Y*. Note that the processor array* R *is accessible via host association.* ■

## 7.2 Dummy Arguments

The distribution of dummy arguments may be specified either *explicitly*, or they may be *inherited*. An explicit distribution is specified in the same way as for non-formal objects. In addition, a dummy array may inherit the distribution from the corresponding actual argument.

The distribution of dummy arrays for which no distribution directive has been specified is determined by the compiler in an implementation dependent way.

### 7.2.1 Dummy Arguments with an Explicit Distribution

For a dummy argument an explicit distribution may be specified in exactly the same way as for the data objects of the main program unit by means of DISTRIBUTE or ALIGN.

An explicit distribution is always enforced on entry to the procedure. As a consequence, if the actual argument does not have a matching distribution, a redistribution must be performed. On exit a similar redistribution may be necessary in order to restore the distribution of the actual argument.

However, if both the actual and the dummy argument have the DYNAMIC attribute, the distribution of the dummy array is returned to the caller without the need for data motion upon procedure exit[8]. Dummy arrays with an explicit distribution are shown in Example 7.1

[8] This is under the assumption that the argument transfer is by reference.

**Example 7.2: Inherit and Range Attribute**

```
SUBROUTINE SUB(X,Y,Z)
   REAL, DIMENSION(:) :: X, Y, Z(:,:)

   !HPF$ INHERIT                    :: X, Y, Z
   !HPF+ RANGE (ALIGN WITH X)       :: Y
   !HPF+ RANGE (BLOCK,*),(*,BLOCK)  :: Z
```

*The inherit and range directives of this example constitute an assertion to the compiler that in all incarnations of procedure* `sub`*,* `X` *and* `Y` *is distributed in exactly the same way, whereas* `Z` *may be partitioned into blocks of columns or blocks of rows, respectively.* ∎

### 7.2.2 Dummy Arguments with an Inherited Distribution

An *inherited* distribution is specified by means of the `INHERIT` attribute. The `INHERIT` attribute specifies that a dummy argument should adopt the distribution of the corresponding actual argument. In this case, the distribution of the actual argument is transferred without the need for moving data[9]. Consequently, an object that has the `INHERIT` attribute must not have the `DYNAMIC` attribute.

Arrays with inherited distributions provide a convenient way to integrate existing (sequential) code into an HPF program. In particular, this mechanism is important for the provision of routines that can be called with arbitrary distributed actual arguments. However, since such dummy arguments may be associated with different distributions, the compiler must parameterize the procedure such that it can accept any incoming distributions. This significantly complicates the task of the compiler and may restrict certain optimizations.

The user may restrict the number of distributions that may be associated with a dummy array that has the `INHERIT` attribute by means of the `RANGE` attribute. The `RANGE` attribute may consist of an arbitrary number of distribution specifications. All these distributions are referred to as *range of permissible distributions*. If the range attribute is specified for a dummy, the actual argument must have a distribution that is contained in its range of permissible distributions. It is illegal to associate such an array with a distribution not contained in the range of permissible distributions. If the `RANGE` attribute is absent, a dummy may be associated with *any* permitted distribution. Example 7.2 shows the use of the `RANGE` attribute for dummy arrays. For a detailed description of the `RANGE` attribute refer to Section 3.6.

If the `RANGE` attribute is specified with a *single* distribution this constitutes an assertion to the compiler that the actual argument will have this distribution in all incarnations of the procedure.

Note, that the HPF+ mechanisms for dummy arrays provide the same functionality as *prescriptive* and *descriptive* mappings of HPF. Moreover, the `INHERIT` attribute in connection with the HPF+ `RANGE` attribute makes *transcriptive mappings*[10] of HPF superfluous.

[9] This is again under the assumption that the argument is passed by reference.
[10] In HPF such mappings are characterized by omitting either the distribution formats or the processor array.

## 7.3 Local Objects of Procedures

For local data objects of procedures a distribution may be specified in the usual way. In particular, such object may be aligned with dummy arguments.

## 7.4 Actions upon Entry and Exit of a Procedure

In this section we take a closer look at the actions that have to be performed upon procedure entry and procedure exit in case of distributed arguments.

### 7.4.1 Procedure Entry

An explicit distribution of a dummy argument is always enforced on entry to a procedure. If the corresponding actual argument does not have an identical distribution, a redistribution according to the explicit distribution directives of the dummy is to be performed by the compiler. This redistribution may be performed by the caller or the callee. No data motion on entry to a procedure is required if the dummy argument has the `INTENT(OUT)` attribute, or if the dummy argument has the `POINTER` attribute[11] and the actual argument is not yet associated with a target or is not yet allocated (see Example 7.3).

### 7.4.2 Procedure Exit

Upon exit from a procedure, the compiler might have to perform a redistribution of a procedure argument if it is necessary to restore the distribution of the actual argument. Redistribution on exit from a procedure is necessary if an actual argument has an explicit distribution that does not match the distribution of the corresponding dummy argument and either the actual array or the dummy array does not have the `DYNAMIC` attribute.

If a dummy has the `INHERIT` attribute or the `INTENT(IN)` attribute, no redistribution upon exit from the procedure will be necessary.

If both the actual as well as the dummy argument have the `DYNAMIC` attribute, no redistribution upon exit from the procedure is performed, and the distribution of the dummy will be returned to the calling procedure (see Example 7.3).

The mechanisms provided by HPF+ for the transfer of distributed objects across procedure boundaries are summarized in Figure 1.

## 7.5 Discussion

HPF+ simplifies the semantics of procedure calls without sacrificing required functionality: the distribution of a dummy argument may either be specified explicitly (in exactly the same way as in HPF) or it may be inherited from the actual argument by using the `INHERIT` attribute possibly combined with the `RANGE` attribute in order to enable the user to provide to the compiler additional information on the number of different distribution that may reach a procedure.

HPF provides four different mechanisms for describing the mapping of dummy arguments, *prescriptive, descriptive, transcriptive* and *inherited* mappings. This relatively large number of different

[11] Note that in this case Fortran requires that the actual argument also has the `POINTER` attribute.

**Example 7.3: Dummy Objects with the `POINTER` attribute**

```
REAL, POINTER, DYNAMIC, DIMENSION(:,:) :: A
...
CALL GET_MEMORY(A)
...
CONTAINS

   SUBROUTINE GET_MEMORY(X)
   REAL, POINTER, DYNAMIC, DIMENSION(:,:) :: X
   !HPF$ DISTRIBUTE(BLOCK,BLOCK) :: X
   ...
      ALLOCATE(X(N,N))
   ...
```

*This example shows how the allocation and distribution of an object can be determined within a procedure. Since both* `A` *and* `X` *have the* `DYNAMIC` *attribute the distribution of* `X` *is transferred back to the calling routine.*

■

mechanisms increases the complexity of the language and its implementation. In particular, transcriptive mappings and the inheritance of templates and/or processor arrays significantly complicate the semantics of the argument transfer. HPF distinguishes between *natural templates* and *inherited templates*, depending on whether the `INHERIT` attribute has been specified. The natural template of a dummy array has always the same shape as the actual array (section), whereas the inherited template is a copy of the template with which the actual array is ultimately aligned, even if the actual array is an array section. Thus, for dummy arguments with the `INHERIT` attribute a descriptive distribution assertion has to be expressed with respect to a template which is not accessible by other objects of the procedure.

Example 7.4 shows how *descriptive* and *transcriptive mappings* of HPF can be expressed in HPF+ by using the `RANGE` attribute. The semantics of `INHERIT` in HPF+ is different from that in HPF. In HPF+ the template of a dummy array always has the same shape as the corresponding actual array section. As long as whole arrays and not array sections are passed to procedures, there is no difference between the HPF inherit attribute and the HPF+ inherit attribute.

Changing the distribution of an object with the `DYNAMIC` attribute as the result of a procedure call has also been adopted by the HPF-2 Approved Extensions.

The `RANGE` attribute, originally introduced in Vienna Fortran, is included within the Approved Extensions of HPF-2. In HPF-2 it is however not possible to specify a *dist-target* (i.e. the processor array to which the distribution refers) or to provide an alignment specification. Furthermore, it is not possible to specify the arguments of the distribution formats. This missing functionality significantly impairs the applicability of the `RANGE` attribute for dummy arguments with the `INHERIT` attribute.

**Case 1: Statically distributed dummy arrays:**

- **Procedure Entry:**
  A redistribution has to be performed on procedure entry iff the distribution of $A$ does not match the distribution of $D$. Physical transfer of data can be omitted, iff $D$ has the `INTENT(OUT)` attribute.

  If $A$ has the `DYNAMIC` attribute and has not yet been associated with a distribution an error message will be issued.

- **Procedure Exit:**
  An implicit redistribution has to be performed on procedure exit iff the distribution of $D$ does not match the distribution of $A$. If $D$ has the `INTENT(IN)` attribute no data transfer is necessary.

**Case 2: Dummy arrays with an inherited distribution**

- **Procedure Entry:**
  A dummy array with an inherited distribution can be associated with any distribution and therefore no redistribution is required on entry.

  If $A$ has the `DYNAMIC` attribute and has not yet been associated with a distribution an error message will be issued.

- **Procedure Exit:**
  Since a dummy array with the `INHERIT` attribute must not not have the `DYNAMIC` attribute its distribution cannot be changed within the scope of the procedure and thus no redistribution is required on exit.

**Case 3: Dynamically distributed dummy arrays**

- **Procedure Entry:**
  A redistribution has to be performed iff $A$ and $D$ have a different distribution. The redistribution can be performed without physical transfer of data, iff $D$ has the `INTENT(OUT)` attribute.

  If $D$ does not have an explicit distribution it adopts the distribution of $A$.

- **Procedure Exit:**

  If $A$ does not have the `DYNAMIC` attribute a redistribution in order to restore $A$'s distribution may have to be performed upon procedure exit.

Figure 1: Transfer of distributed objects across procedure boundaries. $A$ denotes an actual array and $D$ the corresponding dummy array.

**Example 7.4: Mapping of Dummy Arguments - HPF+ versus HPF**

- **Descriptive Mappings:**
  *A descriptive mapping in HPF constitutes an assertion that the mapping of the actual argument matches the mapping specified for the dummy argument. Thus no remapping will be required.*

  ```
  !HPF$ DISTRIBUTE X *(BLOCK) ONTO *R
  ```

  *In HPF+ this can be specified by means of the* `INHERIT` *attribute together with the* `RANGE` *attribute specifying a single distribution.*

  ```
  !HPF+ INHERIT, RANGE (BLOCK) ONTO R :: X
  ```

- **Transcriptive Mappings:**
  *In HPF a mapping is transcriptive, if either the* dist-format-clause *or the* dist-target *consists of an asterisk. This constitutes a request to copy that aspect of the distribution from the actual argument, again without the need for any remapping. The HPF* `INHERIT` *attribute specifies that the dummy argument should be aligned to a copy of the actual argument's template in the same way as the actual argument. In this example the* `INHERIT` *directive has the same meaning as the* `DISTRIBUTE` *statement for* `Z`.

  ```
  !HPF$ DISTRIBUTE X *(BLOCK) ONTO *
  !HPF$ DISTRIBUTE Y * ONTO R
  !HPF$ DISTRIBUTE Z * ONTO *
  !HPF$ INHERIT    Z
  ```

  *In HPF+ such mappings are specified by using the* `INHERIT` *attribute possibly together with a* `RANGE` *attribute that specifies only the distribution or only the distribution target. Note that in HPF+ the* `INHERIT` *attribute has a slightly different semantics, compared to HPF.*

  ```
  !HPF+ INHERIT, RANGE (BLOCK) :: X
  !HPF+ INHERIT, RANGE ONTO R  :: Y
  !HPF+ INHERIT                :: Z
  ```

■

# 8 Task Parallelism

HPF+ as described in the previous sections provides advanced support for the exploitation of data parallelism. However, many applications exhibit multiple levels of parallelism and can exploit some forms of task parallelism in addition to data parallelism. In order to support such applications HPF+ takes over the tasking mechanism as it is specified in the Approved Extensions of HPF-2. We summarize the tasking features of HPF+ below and refer the interested reader to the HPF-2 standard document [14], Section 9, for detailed information.

Task parallelism can be achieved by mapping different computations onto different subsets of a processor array. As a consequence of such a mapping specific statements are executed only on a subset of the available processors rather than on all of them. All processors belonging to a processor subset which executes a specific statement is called the *active processor set* for this statement.

HPF+ provides the following directives for expressing task parallelism:

1. The `ON` directive is used to specify the mapping of computation onto processors.
2. The `RESIDENT` directive asserts that certain data accesses do not require interprocessor data movement beyond the active processor set for their implementation.
3. The `TASK_REGION` construct provides the means to create independent coarse-grain tasks, each of which can itself execute a data-parallel (or nested task-parallel) computation.

## 8.1 The `ON` Directive

The `ON` directive allows the programmer to control the distribution of computation among the processors of a parallel machine. More specifically, the `ON` directive specifies the *active processor set* for a statement or a set of statements.

Two forms of the `ON` directive are provided: a single-statement form and a multi-statement form. Optionally, a `RESIDENT` clause or a `NEW` clause may be specified.

**Syntax: The `ON` Directive**

| | | |
|---|---|---|
| *on-directive* | **is** | `ON` *on-stuff* |
| *on-stuff* | **is** | *home* [`,` *resident-clause*][`,` *new-clause*] |
| *on-construct* | **is** | *block-on-directive*<br>    *block*<br>    *end-on-directive* |
| *block-on-directive* | **is** | `ON` *on-stuff* `BEGIN` |
| *end-on-directive* | **is** | `END ON` |
| *home* | **is** | `HOME` (*variable*) |
| | **or** | (*processors-elmt*) |
| *processors-elmt* | **is** | *processors-name* [ (*section-subscript-list*) ] |

The RESIDENT clause, which will be defined in Section 8.2, asserts that all data accessed by a computation is local to the active processor set. The NEW clause can be used to declare *private* variables as defined in Section 4.

As shown in Example 8.1, ON directives may be nested. In this case, the processors determined by the inner ON must be a subset of the processors of the outer on-block. In other words, the ON directive can shrink the active processor set, but not enlarge it.

**Example 8.1: ON Directive**

```
!HPF$ ON (P(2:4))                 ! P is a processor array
         A(I+1) = A(I) + B(I) + C(I)

      DO I=1,N
!HPF$    ON HOME(A(IDX(I)) BEGIN   ! block-structured variant
            A(INDEX(I)) = B(I)
            B(I) = C(INDEX(I))
!HPF$    END ON
      END DO

!HPF$ ON HOME(X(I,:)) BEGIN       ! nested ON directives
         DO J=1,M
!HPF$       ON HOME(X(I,J))
               X(I,J) = FOO(Y(J,I))
         END DO
!HPF$ END ON
```

■

As with other Fortran compound statements, transfer of control to the interior of an *on-construct* from outside the block is prohibited, because the executable ON directive modifies the active processor set. Analogously, transfers of control from the interior to outside the *on-construct* are also prohibited.

Note that the ON directive only specifies the partitioning of *computation* among processors but does not indicate processors that may be involved in data transfer nor guarantee that its body can be executed in parallel with any other operation.

### ON Directive and Procedures

Whenever a procedure is called from within an ON-block, the callee inherits the active processor set from the caller. In order to be able to explicitly reference the active processor set, a new intrinsic function that returns the number of active processors is introduced.

```
!HPF$ PROCESSORS :: P(NUMBER_OF_ACTIVE_PROCESSORS())
```

Local objects of a procedure without an explicit distribution or with an explicit distribution but no ONTO clause are always distributed to the active processor set. For local objects with an explicit distribution containing an ONTO clause this has to be ensured by the programmer. In case of REDISTRIBUTE the old and new data homes must both be within the active processor set. The same must hold for objects of ALLOCATE statements.

## 8.2 The RESIDENT Clause, Directive, and Construct

The RESIDENT clause is an assertion that the data referenced by a computation are mapped to the active processors. Whether a given data element is resident depends on two facts: where the data is stored and where the computation takes place. Hence, the RESIDENT clause can be added to the ON directive, which is usually the earliest point in the program where the needed facts might be available. Apart from the RESIDENT clause also a RESIDENT directive and construct are provided in order to specify locality assertions to parts of an ON block.

**Syntax: The RESIDENT Clause, Directive, and Construct**

| | | |
|---|---|---|
| *resident-clause* | **is** | RESIDENT *resident-stuff* |
| *resident-stuff* | **is** | [ (*res-object-list*) ] |
| *resident-directive* | **is** | RESIDENT *resident-stuff* |
| *resident-construct* | **is** | *block-resident-directive*<br>*block*<br>*end-resident-directive* |
| *block-resident-directive* | **is** | RESIDENT *resident-stuff* BEGIN |
| *end-resident-directive* | **is** | END RESIDENT |
| *res-object* | **is** | *object* |

While the RESIDENT clause asserts that references to variables within its scope cannot cause data motion beyond the executing processors, data movements may occur within the active processor set[12]. More specifically, the RESIDENT clause asserts that at least *one* copy of an object which is accessed by a statement within the resident scope, and *all* copies of an object which is assigned to by a statement within the resident scope must reside on at least *one* processor of the active processor set.

The resident assertion applies to all references to objects enumerated in the *res-object-list* within in the scope of the RESIDENT clause. If there is no *res-object-list* then *all* references are resident. The RESIDENT directive is similar to the INDEPENDENT directive, in that if it is correct it does not change the meaning of the program. If the RESIDENT assertion is incorrect, the program is not standard-conforming and thus undefined.

### RESIDENT Directives and Procedures

If a RESIDENT directive applies to a procedure reference its semantics differ whether a *res-object-list* is specified or not.

- If a *res-object-list* appears then it is only asserted that the objects within the list are resident for the call, but no assertion is made about the behavior within the called procedure.
- If no *res-object-list* appears then it is asserted that all references in the caller *and the called procedures* are resident.

[12] Note that the semantics of **RESIDENT** is different if it appears as a clause of an INDEPENDENT directive as described in Section 6

**Example 8.2:** `RESIDENT` **Clause**

```
!HPF$ ON HOME (X(I)), RESIDENT(Y(IX(I)))
         X(I) = Y(IX(I)) - Y(IY(I))

!HPF$ ON HOME (IX(J)), RESIDENT(Y)
         X(J) = Y(IX(J)) - Y(J)

!HPF$ ON HOME (IX(K)), RESIDENT
         X(K) = Y(IX(K)) - Y(IY(K))

!HPF$ ON HOME (A(I)), RESIDENT(A(I),B)
         A(I) = F(A(I),B(LO:HI))

!HPF$ ON HOME (A(I)), RESIDENT
         A(I) = F(A(I),B(LO:HI))
```

*For the following discussion we assume that the* `ON` *directive specifies an active processor set containing exactly one processor. Thus, no communication within an active processor set can occur.*

*The first* `RESIDENT` *clause asserts that the first reference to Y is resident, however, the second reference to Y may cause data movement depending on the actual distribution of Y.*

*In the second statement all elements of Y are resident, thus the evaluation of the right-hand-side cannot cause data movement; but no assertion is given about the locality of X.*

*The third clause asserts that all references are resident, thus no data motion is required to execute the statement.*

*The fourth and fifth clauses refer to function references. While the former one only asserts that all arguments are local at the time of the call and nothing is said about the behavior of the function, the latter one asserts that both, the call and the execution of the function cannot cause data movement.* ■

## 8.3 The `TASK_REGION` Construct

Whereas limited forms of task parallelism may be specified by using independent loops and `ON` clauses, *task regions* as outline below provide more flexibility [18].

The `TASK_REGION` directive is used to assert that the block of code within the task region satisfies the following constraints. All outermost `ON` blocks must have a `RESIDENT` attribute. Further, the code inside two such `ON` blocks must not have interfering I/O. Given these constraints, two such `ON` blocks can safely execute concurrently if they execute on disjoint processor subsets. Such an `ON` block is referred to as *lexical task*. Every execution instance of a lexical task is an *execution task* (or simply *task*).

Task regions must be single-entry single-exit code sections and no transfer of control from outside the *task-region-construct* to inside the *task-region-construct* is allowed.

**Syntax: The TASK_REGION Construct**

| | | |
|---|---|---|
| *task-region-construct* | **is** | *block-task-region-directive*<br>*block*<br>*end-task-region-directive* |
| *block-task-region-directive* | **is** | TASK_REGION |
| *end-task-region-directive* | **is** | END TASK_REGION |

The task region is a dynamic concept, i.e. the number of tasks to be generated may be defined at run time. Communication between tasks has to be specified implicitly by means of assignments outside ON-blocks. Explicit communication and synchronization primitives are not provided. In other words, there is no means of data exchange between two concurrent tasks. Outside ON blocks, all processors must follow the same control flow. Exits from a task region (but not from an ON block) are allowed. Nesting of task regions is possible, thus enabling multiple levels of task parallelism.

By means of a task region different task parallel computation models can be expressed. Examples include *parallel sections* as used for multiblock-codes, *nested parallelism* which can be exploited for instance in sorting algorithms, or *data parallel pipelines*.

**Example 8.3: 3-stage Pipeline**

```
      REAL, DIMENSION(N,N) :: A,B,C
!HPF$ PROCESSORS PROC(8)
!HPF$ DISTRIBUTE A(BLOCK,*) ONTO PROC(1:2)
!HPF$ DISTRIBUTE B(BLOCK,*) ONTO PROC(3:4)
!HPF$ DISTRIBUTE C(*,BLOCK) ONTO PROC(5:8)

!HPF$ TASK_REGION
      DO I=1,N
!HPF$    ON (PROC(1:2)), RESIDENT          ! A is resident on proc(1:2)
            CALL READ_IMAGE(A)
         B=A                               ! implicit communication
!HPF$    ON (PROC(3:4)), RESIDENT          ! B is resident on proc(3:4)
             CALL ROW_FFT(B)
         C=B                               ! implicit communication
!HPF$    ON (PROC(5:8)) BEGIN, RESIDENT    ! C is resident on proc(5:8)
              CALL COL_FFT(C)
              CALL WRITE_IMAGE(C)
!HPF$    END ON
      END DO
!HPF$ END TASK_REGION
```

*This example shows how a 3-stage pipeline can be realized by means of a* TASK_REGION *and* ON *blocks.* ∎

### 8.4 Task Parallelism Beyond HPF-2

The *tasking region* concept as described in Section 8 provides only limited support for task parallelism. While it fits for small grained, pipelined tasks, it lacks the capability of coordinating larger modules, because of its restricted inter-task communication and synchronization facilities.

For exploiting coarse grained task parallelism as it can be found for example in multidisciplinary problems, where modules from a variety of related scientific disciplines are to be combined, we propose an extension of Fortran 90, which provides a software layer on top of data parallel languages. This extension is called *ShareD Abstraction (SDA)*. SDAs are the central concept of the Coordination Language Opus [9, 10].

An SDA generalizes Fortran 90 modules by including features from object-oriented data bases and monitors in shared-memory languages. It consists of a set of data structures along with the methods (procedures) which manipulate this data. "Handles" for an SDA may be passed to different components of a program, thus enabling the access to the data and the invocation of methods of this SDA.

An activation of such a method results in spawning a task, thus enabling a parallel execution of the method together with the calling program unit. Tasks are asynchronously executing autonomous activities and have exclusive access to their SDA data. However, if needed, a method may also be called like a normal subroutine, thus blocking the caller's execution.

Condition clauses associated with methods and synchronization facilities embodied in the methods allow the formulation of a range of coordination strategies for tasks. Inter-task communication is possible by accessing and manipulating data of an SDA which is visible to a range of tasks.

A task may provide different levels of parallelism, for example by executing a data parallel HPF program, or by using the fine grained task parallelism of HPF-2. Resources of the system are allocated to an SDA and subsequently to its methods. These resources may be further subdivided for example by using the HPF-2 `ON` directive.

SDAs may be used to embody the functionality of larger modules and to coordinate the parallel execution of such modules. Data exchange between modules can either be realized by passing parameters or by giving different modules access to a common SDA, which then acts as a *data server*. Hence the typical usage of SDAs in multi-disciplinary programs is two-folded: some SDAs act as *data servers* providing the data needed for computation, which takes place at *computation SDAs*.

SDAs therefore provide a high level abstraction for exploiting coarse grained task parallelism between different modules as well as data or fine grained task parallelism within such a module.

## 9 Acknowledgements

The authors would like to thank Guy Lonsdale, Serge Meliciani, George Mozdzynski, Heinz Schiffermüller and the other partners of the HPF+ project for many fruitful discussions on HPF+ language features.

## References


[1] S. Benkner, K. Sanjari, V. Sipkova. VFC - A Compiler for HPF+, Deliverable D2.2c of the ESPRIT IV LTR Project HPF+. University of Vienna, March 1998.

[2] S. Benkner. A User's Guide for HPF+, Deliverable D2.2c of the ESPRIT IV LTR Project HPF+. University of Vienna, March 1998.

[3] S. Benkner, P. Mehrotra, J. Van Rosendale, H. Zima. High-Level Management of Communication Schedules in HPF-like Languages, In *Proceedings of the International Conference on Supercomputing (ICS'98)*, Melbourne, Australia, July 13-17, 1998 (to appear). Also available as: NASA Contractor Report 201740, ICASE Report No. 97-46, NASA Langley Research Center, Hampton, Virginia, September 1997.

[4] S. Benkner. HPF+: High Performance Fortran for Advanced Industrial Applications. Proceedings HPCN'98, Amsterdam, April 1998, Springer Verlag.

[5] S. Benkner, K. Sanjari, V. Sipkova, and B. Velkov. Parallelizing Irregular Applications with the Vienna HPF+ Compiler VFC. Proceedings HPCN'98, Amsterdam, April 1998, Springer Verlag.

[6] S. Benkner and M. Pantano. HPF+: Optimizing HPF for Advanced Applications, Supercomputer Journal, Vol.13, No.2, pages 31-43, 1997.

[7] S.Benkner. Vienna Fortran 90 and its Compilation. Ph.D.Thesis. Technical Report, University of Vienna, Institute for Software Technology and Parallel Systems, September 1994.

[8] S. Benkner, S. Andel, R. Blasko, P. Brezany, A. Celic, B. Chapman, M. Egg, T. Fahringer, J. Hulman, E. Kelc, E. Mehofer, H. Moritsch, M. Paul, K. Sanjari, V. Sipkova, B. Velkov, B. Wender, H.P. Zima.
Vienna Fortran Compilation System. Version 1.2. User's Guide,
Institute for Software Technology and Parallel Systems, February 1996.

[9] B. Chapman, M. Haines, P. Mehrotra, J. Van Rosendale, and H. Zima. OPUS: A Coordination Language for Multidisciplinary Applications. *Scientific Programming*, 1997.

[10] B. Chapman, M. Haines, P. Mehrotra, E. Laure, J. Van Rosendale, and H. Zima. *Opus 1.0 Reference Manual.* Institute for Software Technology and Parallel Systems, University of Vienna, October 1997. (TR 97-13).

[11] L.M. Delves, D. Lloyd, and D. Watson. Parallel Extension for HPF+. Draft HPF+ working paper, March 1996.

[12] H. M. Gerndt. Updating Distributed Variables in SUPERB. Concurrency: Practice and Experience, Vol.2, Sept. 1990.

[13] High Performance Fortran Forum. *High Performance Fortran Language Specification. Version 1.1* TR, Rice University, November 10, 1994.

[14] High Performance Fortran Forum. *High Performance Fortran Language Specification. Version 2.0* TR, Rice University, January 31, 1997.

[15] ISO. Fortran 90 Standard, May 1991, ISO/IEC 1539 :1991 (E)

[16] C.Koelbl, et.al. Reduction Proposal. HPFF working paper. Distributed at HPFF March'96 meeting.

[17] J. J. Nucciarone, Y. Ozyoruk, and L. N. Long. New Life in Dusty Decks: Results of Porting a CM Fortran-based Aeroacoustic Mode l to High Performance Fortran. Proceedings SC97, San Jose, CA, November 1997.

[18] R.Schreiber, et.al. Tasking Proposal. HPFF working paper. Distributed at HPFF March'96 meeting.

[19] V.Sipkova. Constraints on Indirect Distributions and Independent Loops within VFCS 1.2. HPF+ working paper. Distributed at HPF+ Meeting at ESI.

[20] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – a language specification. ICASE Internal Report 21, ICASE, Hampton, VA, 1992.

# A Syntax

This appendix contains the syntax definitions of the HPF+ Language Specification. Note that the syntax of the task parallel features discussed in Section 8 are not included here.

## A.1 Syntax of Directives

| | | |
|---|---|---|
| *hpf-directive-line* | **is** | *directive-origin hpf-directive* |
| *directive-origin* | **is** | `!HPF$` |
| | **or** | `!HPF+` |
| | **or** | `CHPF$` |
| | **or** | `*HPF$` |
| *hpf-directive* | **is** | *specification-directive* |
| | **or** | *executable-directive* |
| *specification-directive* | **is** | *processor-directive* |
| | **or** | *align-directive* |
| | **or** | *distribute-directive* |
| | **or** | *inherit-directive* |
| | **or** | *dynamic-directive* |
| | **or** | *range-directive* |
| | **or** | *halo-directive* |
| | **or** | *shadow-directive* |
| | **or** | *schedule-directive* |
| | **or** | *combined-directive* |
| *executable-directive* | **is** | *independent-directive* |
| | **or** | *redistribute-directive* |
| | **or** | *reset-directive* |

Constraint: An *hpf-directive-line* cannot be commentary following another statement on the same line.

Constraint: A *specification-directive* may appear only where a Fortran *declaration-construct* may appear.

Constraint: An *executable-directive* may appear only where a Fortran *executable-construct* may appear.

Constraint: An *hpf-directive-line* follows the rules of either Fortran free form or fixed form, depending on the source-form of the surrounding program unit.

## A.2 Combined Directives

| | | |
|---|---|---|
| *combined-directive* | **is** | *combined-attribute-list* `::` *combined-decl-list* |
| *combined-attribute* | **is** | `DISTRIBUTE` *dist-attribute-stuff* |
| | **or** | `ALIGN` *align-attribute-stuff* |
| | **or** | `DYNAMIC` |
| | **or** | `INHERIT` |
| | **or** | `PROCESSORS` |
| | **or** | `DIMENSION (` *explicit-shape-spec-list* `)` |
| | **or** | `RANGE` *range-attr-stuff* |
| | **or** | `SHADOW` *shadow-attr-stuff* |
| | **or** | `HALO` *halo-attr-stuff* |
| *combined-decl* | **is** | *hpf-entity* [ `(` *explicit-shape-spec-list* `)` ] |
| | **or** | *entity-decl* |
| | **or** | *object-name* |
| *hpf-entity* | **is** | *processors-name* |

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

Constraint: The `DIMENSION` attribute may appear together only with the `PROCESSORS` directive in a given *combined-directive.*

Constraint: The `DISTRIBUTE` and `ALIGN` attribute must not appear together in a given *combined-directive.*

Constraint: The `DYNAMIC` and `INHERIT` attribute must not appear together in a given *combined-directive.*

## A.3 PROCESSORS Directive

| | | |
|---|---|---|
| *processors-directive* | **is** | `PROCESSORS` *processors-decl-list* |
| *processors-decl* | **is** | *processors-name* [ (*explicit-shape-spec-list*) ] [ `=` *associate-proc-ref* ] |
| *associate-proc-ref* | **is** | *associate-processors-section* |
| | **or** | `RESHAPE (` *associate-processors-section* [`,` *order* ] `)` |
| *associate-processors-section* | **is** | *associate-processors-name* [ `(` *section-subscript-list* `)` ] |
| *order* | **is** | [ `ORDER =`] *order-expr* |
| *order-expr* | **is** | *int-expr* |

Constraint: All *section-subscripts* and *order-expr* must be specification expressions.

Constraint: If *order-expr* is specified, it must be a rank one integer array expression of size equal to the rank of *associate-processors-section.* The value of *order-expr* must be a permutation of `(/1,2,...,n/)`, where `n` is the rank of *associate-processors-section.*

## A.4 DISTRIBUTE and REDISTRIBUTE Directives

| | | |
|---|---|---|
| *distribute-directive* | **is** | `DISTRIBUTE` *distributee dist-attribute-stuff* |
| *redistribute-directive* | **is** | `REDISTRIBUTE` *distributee dist-attribute-stuff* [ , *shadow-or-halo* ] |
| | **or** | `REDISTRIBUTE` *dist-attribute-stuff* [ ,*shadow-or-halo*] :: *distributee-list* |
| *dist-attribute-stuff* | **is** | *dist-format-clause* [ *dist-onto-clause* ] |
| *distributee* | **is** | *object-name* |
| | **or** | *component-name* |
| | **or** | *structure-component* |
| *dist-format-clause* | **is** | ( *dist-format-list* ) |
| | **or** | ( *generalized-dist-format* ) |
| *dist-format* | **is** | `BLOCK` [ ( *int-expr* ) ] |
| | **or** | `CYCLIC` [ ( *int-expr* ) ] |
| | **or** | * |
| | **or** | *extended-dist-format* |
| *extended-dist-format* | **is** | `GEN_BLOCK` ( *int-array-expr* ) |
| | **or** | `MULTI_BLOCK` ( *int-array-expr*, *int-array-expr* ) |
| | **or** | `INDIRECT` ( *int-array-expr* ) |
| *generalized-dist-format* | **is** | `INDIRECT` ( *int-array-expr* ) |
| *int-array-expr* | **is** | *int-expr* |
| *shadow-or-halo* | **is** | `SHADOW` *shadow-attr-stuff* |
| | **or** | `HALO` *halo-attr-stuff* |
| *dist-onto-clause* | **is** | `ONTO` *dist-target* |
| *dist-target* | **is** | *processors-name* |
| | **or** | *extended-dist-target* |
| *extended-dist-target* | **is** | *processors-section* |
| | **or** | `RESHAPE` ( *processors-section* [ , *shape* ] [ , *order* ] ) |
| *processors-section* | **is** | *processors-name* [ ( *section-subscript-list* ) ] |
| *order* | **is** | [ `ORDER =` ] *order-expr* |
| *shape* | **is** | [ `SHAPE =` ] *shape-expr* |
| *order-expr* | **is** | *int-expr* |
| *shape-expr* | **is** | *int-expr* |

Constraint: An *object-name* mentioned as a *distributee* may not appear as an *alignee*.

Constraint: A *distributee* that appears in a `REDISTRIBUTE` directive must have the `DYNAMIC` attribute .

Constraint: The length of the *dist-format-list* must be equal to the rank of each distributee.

Constraint: If both a *dist-format-clause* and a *dist-onto-clause* appear, the number of elements of the *dist-format-list* that are not "`*`" must be equal to or less than the rank of the named processor arrangement of the *dist-onto-clause*.

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

Constraint: A component of a derived type may appear in a `DISTRIBUTE` directive only if its type is not a distributed derived type.

Constraint: An object of a derived type may be explicitly distributed only if the derived type is not an explicitly distributed derived type.

Constraint: A *distributee* in a `DISTRIBUTE` directive must not be a structure component.

Constraint: A *distributee* in a `DISTRIBUTE` directive in a *derived-type-def* must be the *component-name* of a component of the derived type.

Constraint: A *distributee* that is a structure-component may occur only in a `REDISTRIBUTE` directive and every *part-ref* except the rightmost one must be scalar and the rightmost *part-name* must have the `DYNAMIC` attribute.

Constraint: In a *section-subscript-list*, the number of *section-subscripts* must be equal to the rank of *processors-name*.

Constraint: *order-expr* and *shape-expr*, if present in a `REDISTRIBUTE` directive, must be restricted expressions.

Constraint: *order-expr* and *shape-expr*, if present, must be one-dimensional integer expressions of size equal to the rank of *processors-section*.

Constraint: The *int-array-expr* appearing as an argument to `GEN_BLOCK` must be of rank 1 and of size equal to the rank of the corresponding dimension of the target processor arrangement.

Constraint: The sum of the values of the array obtained by evaluating *int-array-expr* of the `GEN_BLOCK` *dist-format* must be equal to the size of the dimension of the array being distributed. All values must be non-negative.

Constraint: Any *int-array-expr* appearing as an argument to `MULTI_BLOCK` must be of rank 1 and of size equal to the rank of the corresponding dimension of the target processor arrangement.

Constraint: Any *int-array-expr* appearing as an argument to `INDIRECT` in a *dist-format* must be a integer array expression of rank 1.

Constraint: The size of *int-array-expr* appearing as an argument to `INDIRECT` in a *dist-format* must be equal to the extent of the corresponding array dimension being distributed.

Constraint: Any *int-array-expr* appearing as an argument to `INDIRECT` in a *generalized-dist-format* must have the same shape as the array being distributed.

## A.5 ALIGN Directive

| | | |
|---|---|---|
| *align-directive* | **is** | `ALIGN` *alignee align-attribute-stuff* |
| *align-attribute-stuff* | **is** | `(` *align-source-list* `)` `WITH` *align-target* `(` *align-subscript-list* `)` |
| | **or** | `WITH` *align-target* |
| *alignee* | **is** | *object-name* |
| | **or** | *component-name* |
| *align-target* | **is** | *object-name* |
| | **or** | *component-name* |
| *align-source* | **is** | *align-dummy* |
| | **or** | `*` |
| *align-dummy* | **is** | *scalar-int-variable* |
| *align-subscript* | **is** | *align-dummy* |
| | **or** | `*` |
| *align-dummy* | **is** | *scalar-int-variable* |

Constraint: An *alignee* must not be a *distributee*.

Constraint: An *alignee* must not have the `INHERIT` attribute.

Constraint: An *align-target* must not have the `OPTIONAL` attribute.

Constraint: If the *align-source-list* is present, its length must be equal to the length of each *alignee* to which it applies.

Constraint: An *align-dummy* must be a named variable, that appears in exactly one dimension of both the *align-source-list* and the *align-subscript-list*.

Constraint: If the *align-target* has the `DYNAMIC` attribute, the *alignee* must also have the `DYNAMIC` attribute.

## A.6 SHADOW Directive

| | | |
|---|---|---|
| *shadow-directive* | **is** | SHADOW *shadow-target shadow-attr-stuff* |
| *shadow-target* | **is** | *object-name* |
| | **or** | *component-name* |
| | **or** | *structure-component* |
| *shadow-attr-stuff* | **is** | ( *shadow-spec-list* ) |
| *shadow-spec* | **is** | *width* |
| | **or** | *low-width* : *high-width* |
| | **or** | * |
| *width* | **is** | *int-expr* |
| *low-width* | **is** | *int-expr* |
| *high-width* | **is** | *int-expr* |

Constraint: Any *int-expr* appearing in a *shadow-spec* of a *specification-directive* must be a *specification expression* with value greater than or equal to zero.

Constraint: The number of shadow-specs must match the rank of the *shadow-target*.

Constraint: A shadow-spec must be "*" if and only if the dist-format of the corresponding array dimension is "*".

## A.7 HALO Directive

| | | |
|---|---|---|
| *halo-directive* | **is** | HALO *halo-target halo-attr-stuff* |
| *halo-target* | **is** | *object-name* |
| | **or** | *component-name* |
| | **or** | *structure-component* |
| *halo-attr-stuff* | **is** | ( *halo-spec-list* ) ON *halo-proc-ref* |
| *halo-spec* | **is** | *int-array-expr* |
| | **or** | * |
| *halo-proc-ref* | **is** | *processors-name* ( *halo-dummy-list* ) |
| *halo-dummy* | **is** | *scalar-int-variable* |

Constraint: The number of *halo-spec*s must match the rank of *halo-target*.

Constraint: A *halo-spec* must be "*" if and only if the *dist-format* of the corresponding array dimension is "*".

Constraint: The *halo-proc-ref* must refer to the same processor array as the corresponding *halo-target* is distributed to.

Constraint: A *halo-spec* must be a rank one integer array expression where all elements have a value greater than or equal to the lower-bound and less-than or equal to the upper-bound of the corresponding dimension of the *halo-target*.

Constraint: Any *int-array-expr* appearing in a *halo-spec* of a *specification-directive* must be a *specification expression.*

Constraint: The number of *halo-dummys* must match the rank of *processors-name*.

## A.8 INHERIT Directive

| | | |
|---|---|---|
| *inherit-directive* | **is** | `INHERIT [ :: ]` *object-name-list* |

Constraint: An object that appears in the *inherit directive* must be a dummy argument.

Constraint: An object that appears in the *inherit directive* must not appear as a distributee in a `DISTRIBUTE` statement, or as an alignee in an `ALIGN` statement and must not have the `DYNAMIC` attribute.

## A.9 DYNAMIC Directive

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

Constraint: An object in `COMMON` may not be declared `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`.

## A.10 RANGE Directive

| | | |
|---|---|---|
| *range-directive* | **is** | `RANGE` *object-name range-attr-stuff* |
| *range-attr-stuff* | **is** | *range-distribution-list* |
| *range-distribution* | **is** | `(` *range-dist-format-list* `) [` *dist-onto-clause* `]` |
| | **or** | `(` *dist-onto-clause* `)` |
| | **or** | `( ALIGN` *align-attribute-stuff* `)` |
| *range-dist-format* | **is** | `BLOCK [ ( [` *int-expr* `] ) ]` |
| | **or** | `CYCLIC [ ( [` *int-expr* `] ) ]` |
| | **or** | `GEN_BLOCK [ (` *int-array-expr* `) ]` |
| | **or** | `MULTI_BLOCK [ (` *int-array-expr*, *int-array-expr* `) ]` |
| | **or** | `INDIRECT [ (` *int-array-expr* `) ]` |
| | **or** | `*` |
| | **or** | `ALL` |

Constraint: An object that has the `RANGE` attribute must have the `DYNAMIC` attribute or the `INHERIT` attribute.

Constraint: The number of *range dist-formats* must be equal to the rank of *object*.

Constraint: Any *int-expr* appearing in a *range-dist-format* attribute must be a specification expression.

Constraint: Any *int-array-expr* appearing in a *range-dist-format* must be a specification expression.

## A.11 INDEPENDENT Directive

| | | |
|---|---|---|
| *independent-directive* | **is** | `INDEPENDENT` [ *independent-clause-list* ] |
| *independent-clause* | **is** | *new-clause* |
| | **or** | *reduction-clause* |
| | **or** | *on-clause* |
| | **or** | *gather-clause* |
| | **or** | *scatter-clause* |
| | **or** | *reuse-clause* |
| | **or** | *resident-clause* |
| *new-clause* | **is** | `NEW` ( *variable-list* ) |
| *reduction-clause* | **is** | `REDUCTION` ( *reduction-variable-list* ) |
| *reduction-variable* | **is** | *array-variable-name* |
| | **or** | *scalar-variable-name* |
| | **or** | *structure-component* |
| *on-clause* | **is** | `ON HOME` ( *variable* ) |
| | **or** | `ON` ( *processors-section* ) |
| *gather-clause* | **is** | `GATHER` (*schedule-def-use-list*) |
| *scatter-clause* | **is** | `SCATTER` (*schedule-def-use-list*) |
| *schedule-def-use* | **is** | *object-list* :: *schedule-variable* [ = *pattern-spec* ] |
| *reuse-clause* | **is** | `REUSE` [ ( [ `COND` = ] *scalar-logical-expr* ) ] |
| *resident-clause* | **is** | `RESIDENT` ( *resident-variable-list* ) |

Constraint: The first non-comment line following an *independent-directive* must be a *do-stmt*, *forall-stmt*, a *forall-construct*, or an array assignment statement.

Constraint: If either the *new-clause* or the *reduction-clause* is present, then the *independent-directive* must apply to a *do-stmt*.

Constraint: A *variable* named in the *new-clause* and any component or element thereof must not:

- Be a dummy argument.
- Have the `SAVE` or `TARGET` attribute.
- Be storage associated with another object.
- Be use associated or host associated.
- Be accessed in another program unit via host association.

Constraint: A *reduction-variable* must be of intrinsic type.

Constraint: A *variable* must not both appear in the *new-clause* and the *reduction* clause of the same *independent* directive.

Constraint: An *variable* referenced in the *on-clause* of an `INDEPENDENT` directive of a *do-stmt* must be an array variable that references in exactly one subscript the loop iteration variable of the *do-stmt*.

Constraint: An *variable* referenced in the *on-clause* of a *forall-stmt* or *forall-construct* must be an array element or section that references all *forall-indices*, each in a different array dimension.

Constraint: An *variable* referenced in the *on-clause* of a array assignment statement must be an array section that is conformable with the left-hand-side array section of the assignment statement.

## A.12 Reduction Statements

| | | |
|---|---|---|
| *reduction-stmt* | **is** | *reduction-var-ref* `=` *expr reduction-op reduction-var-ref* |
| | **or** | *reduction-var-ref* `=` *reduction-var-ref reduction-op expr* |
| | **or** | *reduction-var-ref* `=` *reduction-function*`(`*reduction-var-ref*`,` *expr*`)` |
| | **or** | *reduction-var-ref* `=` *reduction-function*`(`*expr*`,` *reduction-var-ref*`)` |
| *reduction-var-ref* | **is** | *variable* |
| *reduction-op* | **is** | *intrinsic-op* |
| *reduction-function* | **is** | `MAX` |
| | **or** | `MIN` |
| | **or** | `IAND` |
| | **or** | `IOR` |
| | **or** | `IAND` |

Constraint: A reduction variable must not be a substring.

Constraint: The *reduction-var-ref*s in a given *reduction-stmt* must be lexically identical.

Constraint: *expr* must not contain a reference to *reduction-var*.

Constraint: A *reduction-op* must be an associative and commutative operator.

## A.13 SCHEDULE Directive

| | | |
|---|---|---|
| *schedule-directive* | **is** | `SCHEDULE` [ [, *schedule-attr-spec-list* ] `::` ] *schedule-decl-list* |
| *schedule-decl* | **is** | *schedule* [ `(` *schedule-shape-spec-list* `)` ] |
| *schedule-attr-spec* | **is** | `DIMENSION` `(` *schedule-shape-spec-list* `)` |
| | **or** | `SAVE` |
| *schedule* | **is** | *object-name* |
| *schedule-shape-spec* | **is** | *explicit-shape-spec* |

## A.14 RESET Directive

*reset-directive* **is** `RESET` [::] *schedule-variable-list*

## A.15 PUREST Directive

*purest-directive* **is** `PUREST`

Constraint: A *purest-directive* must appear in the specification part of a function or subroutine.