HPF+
Initial Language Definition

Siegfried Benkner, Hans Zima
Institute for Software Technology and Parallel Systems
University of Vienna
Liechtenstein Strasse 22, A-1090 Vienna, Austria

Deliverable: D2.1a
Date: June 1996
Status: draft
Confidentiality: confidential
Responsible: University of Vienna

Revised:

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

©1996 by the **HPF+** Consortium

Contents

1 Introduction **1**
1.1 HPF-1 versus HPF-2 . . . 1
1.2 Summary of HPF+ Features . . . 2

2 Processor Arrays **4**
2.1 Relationship between Processor Arrays . . . 5
2.2 Processor Views and Processor Subgroups . . . 6
2.3 Size of Processor Arrays . . . 6
2.4 Discussion . . . 7

3 Data Distribution **8**
3.1 Overview . . . 8
3.2 Distribution to Subsets of Processors . . . 8
3.3 General Block Distributions . . . 10
3.4 Indirect Distributions . . . 11
3.5 Distribution of Allocatable Arrays and Pointers . . . 15
3.6 Dynamic Data Distribution . . . 15
3.7 Additional Extensions . . . 17

4 Independent Loops **20**
4.1 Syntax of HPF+ Independent Loops . . . 21
4.2 Private Variables . . . 21
4.3 Reduction Operations . . . 21
4.4 Work Distribution . . . 24
4.5 Discussion . . . 26

5 Procedures **30**
5.1 Dummy Arguments with an Explicit Mapping . . . 30
5.2 Dummy Arguments with an Inherited Mapping . . . 30
5.3 Interfaces . . . 32
5.4 Actions upon Entry and Exit of a Procedure . . . 33
5.5 Discussion . . . 34

6 Future Extensions **37**

A Syntax **42**
A.1 Combined Directives . . . 42
A.2 PROCESSORS Directive . . . 42
A.3 DISTRIBUTE and REDISTRIBUTE Directives . . . 42
A.4 RANGE Directive and Attribute . . . 44
A.5 INDEPENDENT Directive . . . 45
A.6 Reduction Statements . . . 45
A.7 INHERIT Directive . . . 46

1 Introduction

In this report we specify the initial language features of HPF+, which is based on Fortran 90/95 and on the current HPF-1 language specification [4]. In order to provide adequate support for the applications considered within the project, the basic data distribution features of HPF-1 have been extended by introducing new mechanisms which are applicable to irregular data structures. The handling of processors has been clarified and generalized in order to facilitate the specification of distributions 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 irregular access patterns. The procedure interface has been significantly simplified without loss of functionality.

Many of the new HPF+ language constructs have been motivated by features provided in Vienna Fortran [9] and Vienna Fortran 90 [1], and have been designed in close connection with existing proposals for HPF-2 [6, 7]. At the time this report is written, a number of HPF-2 features are not yet completely defined and an official HPF-2 language specification is not available. Thus, the differences between HPF+ and HPF-2 are not summarized systematically in this report, but are addressed in the individual sections of this document.

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 current language standard. The HPF Forum is expected to produce a new language version (HPF 2.0) by the end of 1996.

The HPF-2 Language Specification will be structured as follows:

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

Almost all of the new features addressed during the HPF-2 effort (e.g. mapping to processor sections, indirect distributions, derived types, etc.) and a few of the existing HPF 1.1 features that are not yet implemented by commercial HPF compilers (e.g. dynamic redistribution, inheritance of distributions, etc.) will be included only into the Annex of Approved Language Extensions, and are not contained in the core of HPF 2.0. As a consequence, these features will not be supported by early HPF-2 compilers. The HPF-2 core language is essentially a subset of HPF-1, restricted to those features that are already available in most HPF compilers. Thus, it is very unlikely that HPF-2 compiler will provide adequate support for advanced applications like those considered within the HPF+ project.

1.2 Summary of HPF+ Features

The provision of non-standard and irregular distributions, parallel loops with indirect array accesses, and additional support for work distribution and the specification of reduction operations will be fundamental for all HPF+ applications. Other features aim in the correction of current drawbacks and deficiencies of HPF-1, e.g. the handling of processors, argument transfer at procedure boundaries, and others. The following list summarizes the language features available in the initial version of HPF+. All features that have been added extended or modified with respect to HPF 1.1 are marked accordingly. New features that are also addressed in 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 Mapping Mechanisms
  - `BLOCK(M)`, `CYCLIC(M)` distributions
  - Alignment
  - Static Mappings versus Dynamic Mappings
- Extension of Data Mapping Mechanisms
  - *mapping to processor subsets* (new)
  - *general block distributions* (new)
  - *indirect distributions* (new)
  - *mapping of derived type components* (new)
  - *mapping of pointers* (new)
  - *distribution ranges for dynamically mapped objects* (new)
- Independent Loops
  - new-clauses
  - *reduction clauses and reduction statements* (new)
  - on-clauses and distribute-clauses for explicit work distribution (new)
  - *purest procedures* (new)
- Mapping of Dummy Arrays
  - Explicit mappings
  - Inherited mappings (modified)
  - INHERIT attribute (modified)
  - RANGE attribute (new)
  - INTENT attribute

The structure of this report is as follows. Section 2 describes the handling of processor arrays in HPF+. 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, and discusses further possible extensions. Section 5 defines the semantics of argument transfer in HPF+ programs. Finally, in Section 6 additional features addressed in the HPF-2 standardization effort and possible future extensions of HPF+, in particular with respect to task parallelism, are discussed.

2 Processor Arrays

HPF-1 [4] does not assume any relationship between different processor arrays, as long as they do not have the same shape. Within the HPF-2 effort several mechanisms for establishing a relationship between processor arrays, providing different views of a processor array, and language support for defining and naming subgroups of processor arrays have been proposed. However, none of these proposals has been accepted.

As a consequence, HPF programmers may be forced to use templates and alignment whenever arrays of different rank should be distributed to the same processor array[1]. Moreover, a number of quite simple distributions cannot be specified at all. This is illustrated by the HPF+ distributions of Example 2.1 which cannot be expressed in HPF-1.

Example 2.1: HPF+ Distribution not expressible in HPF-1

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

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

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

Since the HPF+ processors declaration specifies a relationship between processor arrays `R1` *and* `R2`*, the above distribution directives specify the following mappings:*

- $A(1:5,1:5)$, $B(1:25)$, *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-1, the distribution for A and B 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 subgroups of processor arrays.

[1] In HPF the number of distributed dimensions (the number of distribution formats that are not '*') of an object 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. A similar proposal has been put forth in [3].

Syntax: HPF+ Processors Directive

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

Constraint: *order-expr* and *shape-expr* must be restricted expressions.

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

If *shape-expr* is omitted, it is as if it were present with `SHAPE=SHAPE(`*processors-name*`)`.

`PRODUCT(`*shape-expr*`)` must be equal to the size of the processor array (section) designated by *associate-processors-section*.

If *order-expr* is specified, it must be a one-dimensional integer 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.

2.1 Relationship between Processor Arrays

HPF+ provides two mechanisms in order to establish a relationship between different processor arrays. The first mechanism, taken from HPF, 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-1 to define a relationship between processor arrays.

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

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* within a processors declaration. The processor array to be declared must have the same shape as the associate processor array (section). An associate processor array may be constructed from an arbitrary section of a previously declared processor array[3].

Note that the implicit relationship between any two processor arrays with identical shape is not established if one of them has been associated with another processor array.

2.2 Processor Views and Processor Subgroups

The intrinsic function `RESHAPE` may be used to specify that a processor array `Q` is a reshape of another previously declared processor array (section) `R`. Correspondence between `Q` and `R` is established according to the Fortran array element order. If the optional *shape* argument is omitted, it is as if it were present with `SHAPE=SHAPE(R)`. The optional *order* argument may be used to specify that the dimension should be permuted, and thus must be a permutation of `(/1,2,...,n/)`, where `n` is the rank of `R`. It can be used to specify arbitrary transpositions of a multidimensional processor array.

Example 2.2: Processor Views and Processor Subgroups

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

The second processors directive declares a processor array `Q` *which corresponds to the same processors as* `R`*. Correspondence is established by means of the Fortran array element order. 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* `QT` *is obtained by transposing* `Q`*. Thus processor* `QT(i,j)` *corresponds to processor* `Q(j,i)`, $\forall i, j, 1 \leq i \leq 2,\ 1 \leq j \leq 4$.

■

2.3 Size of Processor Arrays

According to [4] 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. In order to ensure the portability of codes, HPF+ supports processor arrays of size equal *or less* than the number of physical processors that would be returned by the call `NUMBER_OF_PROCESSORS()`.

[3] A processor array that is referenced as an associate processor 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.

2.4 Discussion

In Vienna Fortran [9] 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 [5]. Furthermore, an implicit one-dimensional processor array is provided. In Vienna Fortran 90 [1], the restriction that all processor arrays must have the same size has been relaxed by allowing procedure local processor arrays to 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, because they also provide support for defining and naming arbitrary subgroups of processor arrays. This significantly improves the flexibility and applicability of the data distribution mechanisms.

3 Data Distribution

3.1 Overview

In this chapter we focus on additional data distribution features required for advanced applications like those considered within the HPF+ project. The basic data mapping mechanisms, e.g. block and cyclic distributions, alignments, and the distinction between statically and dynamically mapped objects are supported in HPF+ in exactly the same way as in HPF-1 and thus are not described in this report.

There are a number of problems, in particular irregular applications that are based on unstructured meshes, for which the distributions available in HPF-1 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. These applications exhibit different levels of parallelism; inter-grid parallelism since usually the computation on each grid may be performed independently of other grids, and intra-grid parallelism which is characterized by the usual 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 suitable sized subsets of processors.

To provide support for such applications HPF+ extends the basic data distribution mechanisms by:

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

All these data distribution extensions have been discussed within the HPF-2 standardization effort. However, none of these features will be included directly in HPF 2.0 but have been accepted for inclusion into the Approved Extensions.

3.2 Distribution to Subsets of Processors

In HPF-1 distributions may refer only to whole processor arrays. Limited forms of mappings to subsets of processors can be achieved in HPF-1 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 for dynamically allocated data objects. In HPF+ it is possible to specify distributions with respect to both regular and irregular subsets of processors.

3.2.1 Regular Processor Array Sections

Distribution to regular processor subsets (i.e. regular processor array sections) is available in Vienna Fortran and has been partially implemented in the VFCS[4]. In HPF+ distributions to regular subsets

[4]Distribution to processor subsets is not yet implemented for `INDIRECT` distributions due to restrictions of the `PARTI` runtime library. See [8] for more details.

of processors may be specified either by using a processor array that has been associated with a regular *subgroup* of another processor array, or by directly specifying a regular processor array section as the target of the distribution (e.g. within the `ONTO clause`). Example 3.1 shows how distributions to regular processor subsets may be defined in HPF+.

The syntax for specifying processor subsets in HPF+ is as follows.

Syntax: HPF+ `ONTO` Clause

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

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

Constraint: If *order-expr* and *shape-expr* of a *dist-target* appear in a `DISTRIBUTE` directive, they must be restricted expressions.

`PRODUCT(`*shape-expr*`)` must be equal to the size of *processors-section*.

If *order-expr* is specified, it must be a one-dimensional integer expression 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 3.1: Distribution to Regular 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 mapped to the first 8 processors of `R`. *B is partitioned into 8 blocks which are mapped to the last 8 processors of* `R`. *Array* `C` *is partitioned into 4 blocks which are mapped 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.* ■

3.2.2 Distribution to Reshaped or Irregular Processor Array Sections

As within a *processors directive*, the `RESHAPE` function may be used to specify arbitrary reshapes of a processor array (section) within the `ONTO` clause. Note that by using the `RESHAPE` function, regular subsets as well as irregular subsets (possibly with permuted dimensions) of a processor array may be referenced. In addition, distributions to irregular processor array sections can be specified by using vector-subscripts[5].

3.3 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:

> Let `A(1:N)` denote a one-dimensional 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)` will be mapped to processor `R(i)`, $\forall i, 1 \leq i \leq$`P`.

Syntax: General Block Distribution

dist-format **is** ...
or `GEN_BLOCK` [(*int-array-expr*)]

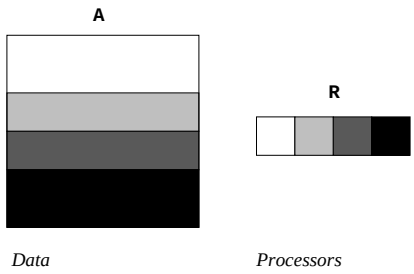
Constraint: Any *int-array-expr* appearing as an argument to `GEN_BLOCK` in the *dist-format* of a `DISTRIBUTE` directive must be a *restricted expression.*

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.

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

[5] In this case, 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.

Example 3.2: 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
```

Array A will be 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, is redistributed by using the array-valued function* `COMPUTE_BLOCK_SIZE()` *as the argument of* `GEN_BLOCK`*.*

■

General block distributions will also be available in the Approved Extensions of HPF-2. The syntax and semantics of the HPF-2 general block distributions is equivalent to HPF+, with the exception that in HPF+ array expressions may be used as arguments of `GEN_BLOCK`, whereas in HPF-2 the argument of `GEN_BLOCK` must always be an array.

3.4 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 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 mapping.

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.

3.4.1 Dimensional Indirect Distributions

A dimensional indirect distribution is used to specify an arbitrary replication-free mapping 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. Note that in the case of an indirect distribution the `ONTO` clause must be present.

Assume that the size of the array dimension to be distributed and the size of the mapping array is given by N, and the size of the corresponding processor array dimension is given by P. Then the i^{th} index of the array dimension is mapped to the j^{th} index 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: HPF+ Dimensional Indirect Distributions

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

Constraint: Any *int-array-expr* apperaing 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.

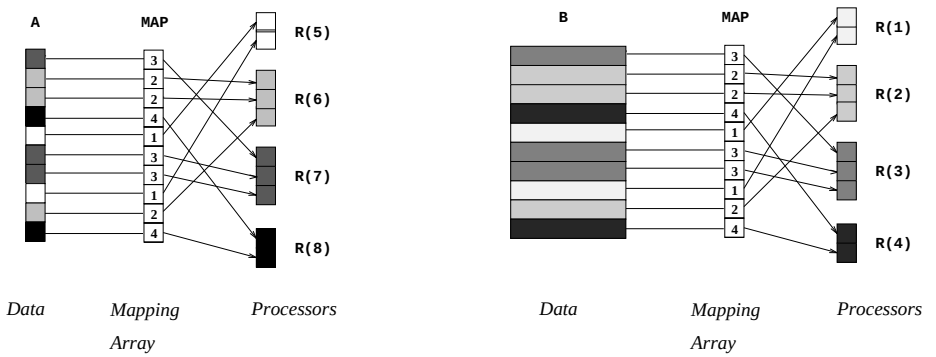
Constraint: The *int-array-expr* appearing as an argument to `INDIRECT` in the *dist-format* of a `DISTRIBUTE` directive must be a restricted expression.

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

Array Expressions as Argument

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 3.3: 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)`. *The* i^{th} *row of* `B`, *i.e.* `B(i,:)`, *is mapped to processor* `R(MAP(I))`. *The dynamically distributed array* `C` *is distributed indirectly by using the array-valued function* `MAP_FUNCTION`. ■

3.4.2 Multidimensional Mapping Arrays

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 [1] 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 mapping (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.

Assume that the size of the array `A` to be distributed and the size of the mapping array `MAP` is 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[6], 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 $1 \leq j \leq P$.

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

Syntax: HPF+ Multidimensional Indirect Distributions

dist-directive-stuff	**is**	...
	or	*generalized-dist-format-clause* [*dist-onto-claus e*]
generalized-dist-format-clause	**is**	(*generalized-dist-format*)
generalized-dist-format	**is**	`INDIRECT` (*int-array-expr*)

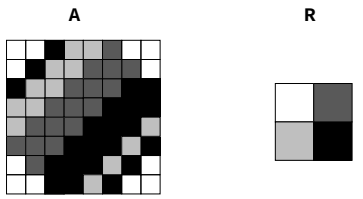
Constraint: Any *int-array-expr* apperaing as an argument to `INDIRECT` in a *generalized-dist-format* must have the same shape as the array being distributed.

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

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

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.

Example 3.4: Multidimensional 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(MAP))
!HPF$ DISTRIBUTE A(INDIRECT(MAP)) ONTO R
```

To specify the above indirect distribution a multi-dimensional mapping array is needed. ■

Whereas dimensional indirect distributions are considered for the Approved Extensions of HPF-2, there will be no support for multidimensional indirect distributions. Note also, that the semantics of the HPF+ `INDIRECT` distribution format is slightly different compared to HPF-2. In HPF+ the values of the mapping array must lie between 1 and the size of the corresponding processor array dimension, whereas in HPF-2 the values of the mapping array must lie between the declared lower-bound and upper-bound of the 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.

3.5 Distribution of Allocatable Arrays and Pointers

Allocatable arrays are handled in HPF+ in exactly the same way as in HPF-1. A variable with the `ALLOCATABLE` attribute may appear as an *alignee* in an `ALIGN` directive or as a *distributee* in a `DISTRIBUTE` directive. Such directives do not take effect immediately, however; they take effect each time the array is allocated by an `ALLOCATE` statement, rather than on entry to the scoping unit. The values of all specification expressions in such a directive are determined once on entry to the scoping unit. For more details on allocatable arrays, please refer to [4].

As opposed to HPF-1, where objects that have the `POINTER` attribute must not be mapped, in HPF+ the mapping of pointer objects may be specified in the same way as for allocatable arrays. Furthermore by using pointers it is possible to hide memory allocation and distribution inside of a procedure. In Fortran, the allocation of an actual array 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 mapping of the actual array after the procedure call will be according to the mapping of the dummy argument. This is described in more detail in Section 5.

3.6 Dynamic Data Distribution

The mapping of an object that has the `DYNAMIC` attribute may be changed within the scope of its declaration by means of the executable directives `REDISTRIBUTE` or `REALIGN`. In order to supply the compiler with information on the set of different distributions a dynamically distributed object may be associated with, the `RANGE` attribute is provided.

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

If the `RANGE` attribute is absent, a dynamically distributed object may be associated with *any* permitted mapping. 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 1.

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 1: Compatibility of *range distribution formats*

Syntax: The RANGE Directive and Attribute

range-directive **is** RANGE *object-name range-mapping-list*

range-attribute **is** RANGE *range-mapping-list*

range-mapping **is** (*range-dist-format-list*) [*dist-onto-clause*]
or *dist-onto-clause*
or *align-attribute-stuff*

range-dist-format **is** BLOCK [([*int-expr*])]
or GEN_BLOCK [(*int-array-expr*)]
or CYCLIC [([*int-expr*])]
or INDIRECT [(*int-array-expr*)]
or *
or ALL

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

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

Constraint: Any *int-expr* appearing in a *range-dist-format* of a dummy object that has the INHERIT attribute must be a specification expression.

Constraint: Any *int-array-expr* appearing in a *range-dist-format* of a dummy object that has the INHERIT attribute must be a restricted expression.

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 3.5 illustrates the use of the RANGE attribute.

Example 3.5: Range Attribute

```
!HPF$ PROCESSORS R(8)
      real :: a(n,n), b(n,n),c(n)
!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 *is either replicated or identically aligned with* B. B *may be partitioned into blocks of columns or blocks of rows, respectively.* C *may be mapped arbitrarily, provided that all mappings are restricted to the first 4 processors of* R. ■

The RANGE attribute may be also specified for dummy arguments with an *inherited mapping* as described in Section 5.

3.7 Additional Extensions

In this section we informally describe some future extensions in the context of data distribution. These features are not yet defined precisely within HPF+ but will be included in the next versions of HPF+.

3.7.1 Mapping of Derived Type Components

In HPF-1, mappings must not be specified for components of a derived type. Mapping of components of derived types has been defined initially in Vienna Fortran 90 based on the concept of *distributed derived types*. A similar proposal has been discussed within the HPF forum and will be included in the Approved Extensions of HPF-2. However, a number of open problems has not yet been solved, in particular, alignment between components of derived type and details in the context of pointers and procedure calls. In the following a brief outline of the proposed HPF-2 mechanisms is given. A detailed specification will be provided in the next version of HPF+.

In HPF-2, a component of a derived type is considered to be mapped if it is either an intrinsic type and explicitly mapped, or it is a structure and any one of its components is mapped.

A component of a derived type may be explicitly mapped if it is of an intrinsic type or if it is a structure and none of its components (at any level of component selection) are mapped. Similarly, a variable of a derived type can be explicitly mapped if none of the components of the derived type are mapped.

Example 3.6 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 3.6: 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 useful for multigrid applications 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`. ■

3.7.2 Shadow Specifications

Shadow specifications (e.g. the width of overlap areas) are proposed for the Approved Extensions of HPF 2.0 in order to provide additional information to the compiler regarding the size of non-local data used during stencil computations, which are common in iterative methods for solving discretized partial differential equations. 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. This additional storage is referred to as *shadow* or *overlap area* and simplifies the organization of communication as well as the local addressing.

The proposed syntax for shadow specifications is as follows:

Syntax: HPF 2.0 Approved Extensions

dist-format	**is**	`BLOCK` [(*int-expr* [, *shadow-spec-or-int-expr*])]
	or	`BLOCK` (*shadow-spec*)
	or	`CYCLIC` (*int-expr*[,*shadow-spec-or-int-expr*])]
	or	...
shadow-spec-or-int-expr	**is**	*shadow-spec*
	or	*int-expr*
shadow-spec	**is**	`SHADOW =` *int-expr*
	or	`LOW_SHADOW =` *int-expr* [,`HIGH_SHADOW =` *int-expr*]
	or	`HIGH_SHADOW =` *int-expr* [,`LOW_SHADOW =` *int-expr*]

For irregular applications, such as the ones considered in HPF+, the situation is more difficult. Since these applications are based on irregular distributions and access patterns, the shadow-concept as

proposed in HPF-2 will not be sufficient, since it is restricted to regular accesses and nearest neighbor communication. It may be therefore necessary that the user specifies non-local accesses, referred to as *halo* or *irregular overlap*, explicitly.

4 Independent 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. Each of the three HPF+ benchmark codes contains indirect addressing: in PAM-CRASH and FIRE it is seen in the unstructured mesh, for the IFS the Semi-Langrangian advection uses a lookup table for accessing remote data.

HPF-1 provides the `INDEPENDENT` directive which enables the programmer to assert that the iterations of the immediately following `DO` loop (or `FORALL`) do not *interfere*[7] and thus may be executed in *any* order. Furthermore, the optional `NEW` clause can be used to declare *private* variables, which can be used as temporaries within the loop. Variables specified in the `NEW` clause are conceptually local in each iteration, and therefore cannot cause loop-carried dependences.

However, as can be seen from Example 4.1 which shows a simplified excerpt of the proposed first version of the PAM-CRASH kernel, additional language support is needed. The reduction operations performed on array `A` cause loop-carried dependencies and thus the use of an `INDEPENDENT` directive would be non-conforming to the rules of HPF-1. In order to use the HPF-1 independent directive the programmer would be forced to expand array `A` by adding a third dimension of size equal to the number of loop iterations (i.e. `numel`). Therefore, additional language support for specifying reduction operations is crucial.

Example 4.1: Simplified Excerpt from PAM-CRASH Kernel

```
real, dimension(6,numnp) :: a
real, dimension(4,numnp) :: ix
real, dimension(6,numel) :: force1,force2,...

   do iel = 1,numel
      nn1 = ix(1,iel)
      nn2 = ix(2,iel)
      ...
      a(nn1) = a(nn1) + force1(iel)
      a(nn2) = a(nn2) + force2(iel)
      ...
   enddo
   ...
```

The indirect array accesses and the use of the scalar variables `nn1, nn2` *which are used to minimize the access to the indirection array* `ix` *could be handled with an* `INDEPENDENT` *directive and* `NEW` *clauses. However, the reduction operations on array* `A` *will cause interference of the loop iterations and thus prohibit the use of the HPF-1* `INDEPENDENT` *directive.*

■

[7] For a definition of interference see [4] pages 84-85.

Besides the lack of support for reduction operations, which often account for a significant amount of the overall computation in irregular applications, a proper distribution of the loop iterations is crucial for minimizing accesses to non-local data. Therefore, the user should be able to provide the compiler with additional information on how the loop iteration space should be mapped to the available processors. In Vienna Fortran this is achieved by providing an explicit work-distribution specification for independent loops using an *on-clause*.

In HPF+, the current deficiencies of the HPF-1 independent loop concept are addressed by additionally providing support for reduction operations and for specifying the distribution of loop iterations (i.e. work distribution). The syntax and semantics of HPF+ independent directives is specified in the next Section.

4.1 Syntax of HPF+ Independent Loops

In HPF+ the `INDEPENDENT` directive (including the *new-clause*) can be used with exactly the same syntax and semantics as in HPF-1. The extensions, described below provide additional support for reduction operations and for specifying the distribution of loop iterations (i.e. work distribution).

4.2 Private Variables

Private variables of an independent loop which may be used as temporaries can be declared by means of the `NEW` clause. Variables specified in the `NEW` clause are conceptually local in each iteration, and therefore cannot 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.

4.3 Reduction Operations

The HPF+ extensions for the support of reduction operations in independent loops are based on the language features currently discussed within the HPF Forum for inclusion into the Approved Extensions of HPF 2.0.

4.3.1 `REDUCTION` Clause

Reduction operations within independent loops may be specified by using a `REDUCTION` clause within the *independent-directive*. As an additional exception to the HPF-1 definition of *interference*[8] permitted references to reduction variables do not cause interference of loop iterations.

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 (e.g. associative operation). In particular, it may not occur in any directive including the variable list in a

[8] See HPF-1.1 specification page 84.

`NEW` clause. This includes any `NEW` clause in the same directive.

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.

Syntax: HPF+ Independent Loops

independent-directive	**is**	`INDEPENDENT` [*independent-clause-list*]
independent-clause	**is**	*new-clause*
	or	*reduction-clause*
	or	*work-distribution-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*
work-distribution-clause	**is**	*on-clause*
	or	*distribute-clause*
on-clause	**is**	`ON HOME` (*array-element*)
	or	`ON` (*processor-array-element*)
distribute-clause	**is**	`DISTRIBUTE` (*dist-format*) [*dist-onto-clause*]

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

Constraint: If one of the *independent-clause*s is present, then the directive must apply to a `DO` loop.

Constraint: A *variable* named in the `NEW` option or any component or element thereof must not:

- Be a pointer or dummy argument; nor
- Have the `SAVE` or `TARGET` attribute.

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

Constraint: A *processors-section* appearing in an *dist-onto-clause* of a *distribute-clause* must have rank 1.

4.3.2 Reduction Statements

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

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 two *reduction-var-ref*s in a reduction statement 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).

Example 4.2: Reduction Directive

```
real F(N),IDX(N),X(M)

!HPF$ INDEPENDENT, REDUCTION(X)
do i=1,n
   x(idx(i)) = x(idx(i)) + f(i)
end do
```

■

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

4.4 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. Moreover, a *distribute-clause* may be used to explicitly specify a distribution of loop iterations. This mechanism, also referred to as *work distribution*, is at the iteration level, i.e., full iterations may be assigned to a single processor.

4.4.1 On Clause

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 4.3: `ON`-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 will 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.*

■

4.4.2 Distribution Clause

In addition to the *on-clause*, the user may specify a work distribution by directly distributing the loop iteration space to a processor array using the same mechanisms as provided for data distribution. Formally, a work distribution can be defined as a total function $\varpi : \mathbf{I}^{\mathcal{L}} \rightarrow \mathbf{I}^{R}$ from the loop iteration space represented by the index domain $\mathbf{I}^{\mathcal{L}} = [1 : N]$ to the index domain of a processor array, denoted by $\mathbf{J}^{R} = [1 : P]$.

Assume that the total number of loop iterations is given by N, and that the index domain of the processor array to which the distribution of loop iterations refers is given by $[1, P]$. Depending on the specified *distribution format* the loop iterations are assigned to the processors as follows:

- `BLOCK(M)`
  If the block size M is specified explicitly, the restriction $M \geq \left\lceil \frac{N}{P} \right\rceil$ must always be true. If the optional block-size M is omitted, it is if were present with $M = \left\lceil \frac{N}{P} \right\rceil$. The N loop iterations are divided into $b = \lfloor \frac{n}{m} \rfloor$ contiguous blocks of size M which are mapped 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 $(N \ mod \ M)$ iterations which is assigned to processor $b + 1$. All other processors (if any) do not receive any iterations. The work distribution function is given by

$$\varpi(i) = \left\{ \left\lceil \frac{i}{M} \right\rceil \mid 1 \leq i \leq N \quad \wedge \quad M \geq \left\lceil \frac{N}{P} \right\rceil \right\}$$

- `GEN_BLOCK(B)`
  In this case, array B must be a one-dimensional integer array of size P. Furthermore, each element of B must be non-negative and `SUM(B) = N` must be satisfied. The general block distribution partitions the loop iterations into P blocks. Block k of size $B(k)$ will be mapped to processor k, $\forall k, 1 \leq k \leq P$. The work distribution function is given by[9]:

$$\varpi(i) = \{k \mid \texttt{SUM}(B(1 : k-1)) < i \leq \texttt{SUM}(B(1 : k)), 1 \leq k \leq P\}$$

- `CYCLIC(M)`
  In this case, the loop iteration space is distributed into chunks of size `M` which are mapped in a round-robin fashion to the processors. If M is omitted, it is as if it were present with $M = 1$. The work distribution function is given by:

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

- `INDIRECT(MAP)`
  In this case, the loop iteration space is distributed according to the contents of the one-dimensional integer array `MAP`. The size of `MAP` must equal the number of loop iterations[10]. The i^{th} loop iteration is assigned to processor `R(MAP(i))`. Thus, the work distribution function is given by:

$$\varpi(i) = \{\texttt{MAP}(i) \mid \ 1 \leq i \leq N\}$$

[9] We assume that the lower bound of B is 1.

[10] For simplicity we assume that the mapping array is declared with lower bound 1

Example 4.4: Distribute-Clause

```
!HPF$ PROCESSORS R(10)

!HPF+ INDEPENDENT, DISTRIBUTE(BLOCK) ONTO R
do k=1,100,2
   ....
enddo
```

In this example the total number of loop iterations is 50. For example, processor `R(1)` *will execute the loop iterations with the values of the iteration variable* $k = 1, 3, 5, 7, 9$. ■

4.5 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.

The additional language features for independent loops 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 [2]. For specifying reduction operations, a new executable statement is provided which in addition to intrinsic reduction operations also allows user-defined reductions.

For HPF-2, the `ON HOME` directive has been proposed which can be used by the programmer to recommend where operations should be executed. The `ON HOME` directive is more general than the `ON` clause of Vienna Fortran. It has been primarily designed in order to provide support for the specification of task parallelism. A free-standing `ON HOME` directive, which applies to the immediately following statement, as well as a block-structured variant are provided. The final syntax and semantics are still undefined. It is however clear, that `ON HOME` and `ON` directives will be not in the core HPF-2 language, but in the Approved Extensions.

The syntax and semantics of HPF+ independent loops is in correspondence with the language features under consideration in HPF-2 with the exception that an additional *work-distribution-clause* may be specified. In addition to the *on-clause* which is similar to the free-standing `ON` directive proposed for HPF-2, HPF+ provides a *distribute clause* to specify a mapping of loop iterations to processors using the same mechanisms as for the specification of data distribution. Furthermore in HPF+, new-clause, reduction-clause, on-clause, and distribute-clause may be specified in any order.

A prototype implementation of `INDEPENDENT` loops that supports indirect array accesses, *on-clauses*, and *work-distribution clauses*, is available in the Vienna Fortran Compilation System (VFCS). The current restrictions of this implementation are described in detail in [8].

4.5.1 Possible Future Extensions of Independent Loops in HPF+

In this section we informally describe some future extensions in the context of independent loops. These features are not yet defined precisely within HPF+ but may be considered for later versions.

Invariant Directive

An *invariance*-directive, of the form

```
!HPF+ INVARIANT(IDX,IDY)
```

before a do loop asserts to the compiler that the listed variables will not be modified within the loop. This information is useful to determine whether certain communication patterns that are dependent on these variables may be changed within the loop. If not, the compiler can arrange the reuse of these communication patterns.

Purest Procedures

HPF-1 provides 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 called from within a `FORALL` loop or construct. Pure procedures must obey a number of syntactic constraints which have to be checked by the compiler. Pure procedures will be also available in the next Fortran standard (i.e. Fortran 95) which is expected in fall '96.

Besides many other constraints, a pure procedure is not allowed to perform any remapping of global data or local data objects. As a consequence, dummy arguments may not appear in mapping directives that fixes 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 be explicitly distributed or given the `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. For example, in the `PAM-CRASH` kernel it will be necessary to call procedures that exclusively work on *private* data objects.

For an efficient implementation of `INDEPENDENT` loops that access distributed objects irregularly 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 requirement that its invocation does not require communication, neither directly nor indirectly. More precisely, the invocation of a purest procedure must not result in communication on entry, during activation, and upon exit from the procedure. The `PUREST` attribute attribute is specified in the *function-stmt* or *subroutine-stmt*.

Procedures without Communication

In some cases the restrictions of *purest* procedures may be too stringent. For example, it is not possible to call a procedure that does not require any communication but is not *pure* (i.e. side effect-free) from an independent loop, as shown in the example below.

Example 4.5: Silent Procedures

```
!HPF$ PROCESSORS R(4)
      REAL,DIMENSION(N) :: GLOBAL_VARIABLE
!HPF$ DISTRIBUTE(BLOCK) :: GLOBAL_VARIABLE

!HPF+ INDEPENDENT, NEW(LX,LY,LZ), ON HOME(GLOBAL_VARIABLE)
      DO I=1,N
         LX = ....; LY = ...; LZ = ...;  ! private variables
         CALL SUB(LX,LY,LZ,I)
         ...
      END DO

!HPF+ SILENT
      SUBROUTINE SUB(X,Y,Z,I)
      ...
         GLOBAL_VARIABLE(I) = ....
      ...
      END SUBROUTINE SUB
```

In this example the procedure SUB *is called with arguments that are local to each loop iteration. Within* SUB *a distributed global variable is assigned and thus the procedure cannot be declared as* PURE *or* PUREST. *However, since the user asserts via the* SILENT *directive that the invocation of this procedure does not result in any communication, the loop can be executed in parallel.* ■

Schedule Reuse

In VFCS, the current prototype implementation of independent loops with indirect data accesses is based on the inspector/executor paradigm supported by the PARTI/CHAOS runtime libraries. With this approach the compiler translates an independent loop into an inspector phase and an executor phase. During the inspector phase the communication schedule to implement non-local data accesses (including storage requirements for non-local data) is determined. This information is then used in the executor phase which performs the actual computation.

The goal here is to reuse the communication schedule as long as the access pattern of the loop does not change in order to amortize the overhead of the inspector. For example, if there is an outer time-step loop and the access pattern is invariant over time, then this approach will work with reasonable efficiency.

However, it may be impossible for the compiler to determine the invariance of communication schedules automatically, and thus additional language support is needed. The following example shows how such additional language support could be provided.

Example 4.6: Schedule Reuse Directive

```
!HPF+ SCHEDULE(A,B)
DO t=1,max_time
   !HPF+ INDEPENDENT, REUSE(SCHEDULE(A,B))
   DO i=1,N
      B(...) = A(idx(i)) ....
   END DO
   ....
   !HPF+ INDEPENDENT, REUSE(SCHEDULE(A))
   DO i=1,N
      ... = A(idx(i)) ...
   END DO
END DO
```

`SCHEDULE(array-list)` *recommends where the inspector should be placed.* `REUSE(SCHEDULE(array-list))` *asserts to the compiler that the previously generated schedule(s) can be reused.* ■

Another approach could be the provision of user-defined functions to determine a communication schedule either within an array declaration or within loops.

5 Procedures

The mappings of dummy arguments may be specified either *explicitly*, or they may be *inherited*. An explicit mapping is specified in the same way as for non-formal objects (i.e. using `ALIGN` or `DISTRIBUTE`). In addition, a dummy array may inherit the mapping from the corresponding actual argument. For arrays with an *inherited mapping*, the *range attribute* may be provided in order to supply the compiler with additional information on incoming distributions.

The mapping of dummy arrays that have neither an explicit mapping nor an inherited mapping is determined by the compiler in an implementation dependent way. For local data objects of procedures mappings may be specified in the usual way. In particular, such object may be aligned with dummy arguments.

Usually any remapping of arguments that might be necessary on entry or upon exit from a procedure is not visible to the caller. Upon return from a procedure all actual arguments are mapped in exactly the same way as they were before the call. This is exactly the same behavior as in HPF-1. However, if both the actual argument and the corresponding formal argument have the `POINTER` as well as the `DYNAMIC` attribute, then the mapping of the actual argument may be changed as the result of a procedure call. Note, however that this mechanism should be used carefully since it significantly complicates the compiler's task to determine which mappings may *reach* a certain reference in the code. This is discussed in more detail in Section 5.4.

5.1 Dummy Arguments with an Explicit Mapping

For a dummy argument an explicit mapping may be specified in exactly the same way as for the data objects of the main program unit. An explicit mapping is enforced on entry to the procedure. As a consequence, if the actual argument does not have a matching mapping, it will be remapped accordingly.

As already mentioned, the mapping of an object does not change as the result of a procedure call as long as it does not have the `POINTER` and `DYNAMIC` attribute. Under this assumption, any remapping that might be necessary on entry or upon exit from a procedure is not visible to the caller. Thus, upon exit from a procedure it may be necessary to restore the original mapping.

It is recommended to specify for all dummy arguments the `INTENT` attribute. This information can be exploited by the compiler to avoid moving data on entry or upon exit from a procedure. 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. Dummy arrays with an explicit mapping are shown in Example 5.1

5.2 Dummy Arguments with an Inherited Mapping

An *inherited* mapping is specified by means of the `INHERIT` attribute. The `INHERIT` attribute specifies that a dummy argument should have exactly the same mapping as the corresponding actual argument. In this case, the mapping of the actual argument is transferred into the procedure without the need for

Example 5.1: Explicitly Mapped 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. On the contrary, 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.* ■

moving data[11]. Consequently, an object that has the `INHERIT` attribute must not have the `DYNAMIC` attribute.

Arrays with inherited mappings 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 mapped actual arguments. However, since such dummy arguments may be associated with different mappings, the compiler must parameterize the procedure such that it can accept any incoming mappings. This significantly complicates the task of the compiler and may restrict certain optimizations.

The user may restrict the number of mappings 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 mapping specifications. All these mappings are referred to as *range of permissible mappings*. If the range attribute is specified for a dummy, the actual argument must have a mapping that is contained in its range of permissible mappings. It is illegal to associate such an array with a mapping not contained in the range of permissible mappings. If the `RANGE` attribute is absent, a dummy may be associated with *any* permitted mapping. For a detailed description of the `RANGE` attribute refer to Section 3.6.

If the `RANGE` attribute with a *single* mapping is specified for a dummy array with an inherited

[11] This is under the assumption that the argument is passed by reference.

Example 5.2: Inherit and Range Attribute

```
subroutine sub(x,y,z)
real, dimension(:) :: x,y,z(:,:)
!HPF+ INHERIT, RANGE (BLOCK) ONTO R      :: X
!HPF+ INHERIT, RANGE ALIGN WITH X        :: Y
!HPF+ INHERIT, 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` *will be distributed in exactly the same way, whereas* `Z` *may be partitioned into blocks of columns or blocks of rows, respectively.* ■

distribution, this constitutes an assertion to the compiler that the actual argument will have only this corresponding mapping in all incarnations of the procedure. Note, that this mechanism provides the same functionality as *descriptive* mappings of HPF-1. Moreover, the HPF+ `RANGE` attribute provides the functionality in order to specify *transcriptive mappings* of HPF-1. In HPF-1 such mappings are characterized by omitting either the distribution formats or the processor array.

5.3 Interfaces

As in HPF, 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 remapping when a procedure is called, the procedure's interface must be explicit[12] 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 the result is array valued or a pointer.
- The procedure has a dummy argument with an *inherited* mapping.
- The procedure has a dummy argument with an *explicit* mapping, which, in at least on incarnation of the procedure, might be different from the mapping of the corresponding actual argument.

If the interface of a procedure is not explicit to the caller, no remapping of actual arguments can occur when the procedure is called.

If the mapping of the dummy arguments is known within the calling routine, then the compiler will arrange for remapping the actual argument to the specified distribution (and possibly mapping it back on return from the subprogram, if necessary). As already mentioned, the provision of an intent specification can help the compiler to reduce data movements at procedure boundaries.

[12] 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.

5.4 Actions upon Entry and Exit of a Procedure

Procedure Entry

The explicit distribution of a dummy argument is always enforced on entry to a procedure. If the corresponding actual argument does not have the same mapping, a redistribution according to the explicit mapping directives of the dummy will be performed by the compiler. In most cases this redistribution will be performed by the caller which implies that the interface of the procedure must be explicit.

However, if the dummy argument has the `POINTER` attribute[13] and the actual argument is not yet associated with a target (i.e. not yet allocated), then the mapping of the dummy argument cannot be determined by the caller but has to be delayed until the dummy gets associated (see Example 5.3).

Procedure Exit

Upon exit from a procedure, the compiler might have to perform a remapping of a procedure argument if it is necessary to restore the distribution of the actual argument. Remapping on exit from a procedure is necessary if an actual argument has an explicit mapping that does not match the mapping of the corresponding actual array and the dummy array does not have both the `POINTER` and `DYNAMIC` attribute. If a dummy has the `INHERIT` attribute or the `INTENT(IN)` attribute, no remapping upon exit from the procedure will be necessary.

If both the actual as well as the dummy argument have the `POINTER` and `DYNAMIC` attribute, no remapping upon exit from the procedure will be performed. As a consequence, the mapping of the dummy will be carried back to the calling procedure (see Example 5.3).

Example 5.3: 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 mapping of an object can be determined within a procedure. Since `A` *and* `X` *have both the* `POINTER` *and* `DYNAMIC` *attribute the distribution of* `X` *will be transferred back to the calling routine.* ■

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

5.5 Discussion

In HPF+ the mapping 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 distributions that may reach a procedure.

HPF-1 provides four different mechanisms for describing the mapping of dummy arguments, *prescriptive, descriptive, transcriptive* and *inherited* mapping directives. The relatively large number of different mechanisms increases the complexity of both the language and the implementation. In particular, transcriptive mappings and the inheritance of templates and/or processor arrays significantly complicate the semantics of the argument transfer.

HPF makes a distinction 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. This however may not be possible due to syntactic constraints as outline in Example 5.4.

Example 5.4: HPF-1 `INHERIT` Attribute

```
      real a(128), b(128), c(128,128)
!HPF$ DISTRIBUTE (BLOCK)       :: A, B
!HPF$ DISTRIBUTE (BLOCK,BLOCK) :: C
      ...
      call sub(a(1:4:3), b(1:4:3), c(1:4:3,1))
      ...
      subroutine sub(x,y,z)
      real, dimension(:) :: x,y,z
!HPF$ INHERIT X,Z
!HPF$ DISTRIBUTE *(BLOCK)       :: X
!HPF$ DISTRIBUTE *(BLOCK)       :: Y  ! illegal
!HPF$ DISTRIBUTE *(BLOCK,BLOCK) :: Z  ! contradictory
```

In this example the descriptive assertion for `A` *is correct since the template of* `X` *is inherited and thus a copy of the template of* `A` *with shape* `[128]`*. The descriptive assertion for* `B` *is illegal since the template of* `B` *is the* natural template *with shape* `[14]`*. The distribute directive of* `Z` *would be correct since the template of* `Z` *has shape* `[128,128]` *but is illegal because the syntactic constraint that the length of a* dist-format-list *must match the rank of each distributee is not fulfilled.* ■

The mechanisms provided for the mapping of dummy arguments in HPF+ have been motivated by those available in Vienna Fortran. We have reduced the number of different mechanisms for the mapping of dummy arguments to two, compared to four in HPF. This significantly simplifies the semantics of procedure calls without sacrificing required functionality.

Example 5.5: 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. In particular, it does not allow the user to infer from it any assumptions about the template of the actual argument.*

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

■

Example 5.5 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 to 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.

The `RANGE` attribute, originally introduced in Vienna Fortran, will be 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.

6 Future Extensions

There are a number of issues which have not yet been addressed or fully specified in this initial definition of HPF+, but will be considered for future versions.

Open Issues:

- support for irregular shadow areas
- support for inspector placement and schedule reuse
- restrictions on procedures called from independent loops
- nesting of parallel constructs, in particular of `INDEPENDENT` loops
- user-defined distributions
- task parallel extensions

In the following, we give a brief overview of features that are currently discussed within the HPF Forum, and which could serve as a basis for future HPF+ extensions, in particular with respect to task parallelism. This includes the `ON HOME` directive, `RESIDENT` directives, and `TASKING` regions. So far, none of these features is completely specified. They are included here in order to provide a basis for discussion on whether such language extensions would be useful within the HPF+ project.

HPF-2 `ON HOME` and `ON` Directive

The `ON HOME` and `ON` directive recommends where operations should be executed. The primary purpose of these directives is to allow the programmer to control the distribution of computations among the processors of a parallel machine. Besides this, they may be used to identify data accessed by a computation as local to the executing processor(s) by providing the `RESIDENT` clause.

An `ON` directive may be specified in single-statement form or as a block-construct. Optionally, the `RESIDENT` clause or the `NEW` clause may be specified. Declarative `ON` directives for procedures have been dropped.

Two forms of the ON-directive are provided:

```
ON HOME(variable)
ON (proc-array-section)
```

Note that the first form of the `ON`-directive may result in an irregular processor array section. Consequently, irregular processor array sections may also be specified explicitly within the second variant.

As shown in Example 6.1, `ON HOME` directives may be nested. In this case, the processors determined by the inner `ON HOME` must be a subset of the processors of the outer on-block.

Example 6.1: `ON HOME` Directive

```
!HPF$ ON HOME(a(i))
      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 HOME directives
do j=1,m
   !HPF$ ON HOME(x(i,j))
      x(i,j) = foo(y(j,i))
end do
!HPF$ END ON
```

■

ON-Directive and Procedures

Whenever a procedure is called from within an `ON`-block, the callee inherits the set of active processors from the caller. In order to be able to explicitly reference the set of active processors, a new attribute for processor declaration statements and a new intrinsic function that returns the number of active processors are proposed.

```
PROCESSORS, ACTIVE :: P(NUMBER_OF_ACTIVE_PROCESSORS())
```

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

The `ON HOME` directive is also used in the context of task parallelism as discussed later.

Locality Assertions

In HPF-2 the `RESIDENT` clause, which is appended to the `ON` directive is proposed. It asserts that the data referenced in the `RESIDENT` clause is local on the executing processor(s) which are specified by the `ON` directive. Thus no data movements beyond the executing processors are necessary. However, data movements may occur within the processors designated by the `ON` directive!

Several forms of the resident directive are under consideration. This includes the possibility of specifying `RESIDENT` within an `ON` directive, and a block-structured version.

A different proposal for RESIDENT is discussed in [3].

Example 6.2: RESIDENT **Directive**

```
!HPF$ ALIGN y(i) WITH x(i)
!HPF$ ON  (procs(1:2)), RESIDENT(x) ! every element of x touched is local
        x(i) = x(indx(i))           ! i, indx1(i), x(indx1(i)) are local
        y(i) = y(indx2(i))          ! y(i) local
!HPF$ END ON


!HPF$ ON HOME(indx1(i)) BEGIN
        n1 = indx1(i)
        n2 = indx2(i)
        !HPF$ RESIDENT (x(n1), y(n1)) BEGIN  ! block-structured variant
           tmp = y(n1) - y(n2)
           x(n1) = x(n1) + tmp
           x(n2) = x(n2) - tmp
        !HPF$ END RESIDENT
!HPF$ END ON
```

■

Tasking Regions

Whereas limited forms of task parallelism may be specified by using independent loops and ON HOME directives, *tasking regions* as outline below provide more flexibility [7].

Tasking regions must be single-entry single-exit code sections that may contain multiple tasks which are delimited by ON directives. The ON directives are used to specify named subgroups of processors which will execute the enclosed statements, which are also referred to as *on-blocks*.

The TASKING directive implies the RESIDENT attribute for each on-block (i.e. all accessed variables are local to the executing processors). By specifying a task region that contains several on-blocks that work an disjoint processor subsets, the user asserts that there is no conflicting I/O between these on-blocks. The NEW clause may be used to specify *private* variables.

This proposal 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. Outside ON blocks, all processors must follow the same control flow. Exists from a tasking region (but not from an ON block) are allowed but may inhibit parallelism. Nesting of TASKING regions will be possible.

Example 6.3: 3-stage Pipeline

```
!HPF$ TASKING
      do i=1,n
         !HPF$ ON HOME (PROC(1:2)) BEGIN   ! A is resident on proc(1:2)
               CALL READ_IMAGE(A)
         !HPF$ END
         B=A                               ! implicit communication
         !HPF$ ON HOME (PROC(3:6)) BEGIN   ! B is resident on proc(3:6)
               CALL ROW_FFT(B)
         !HPF$ END
         C=B                               ! implicit communication
         !HPF$ ON HOME(PROC(7:8)) BEGIN    ! C is resident on proc(7:8)
               CALL WRITE_IMAGE(C)
         !HPF$ END
      end do
!HPF$ END TASKING
```

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

■

The syntax and semantics of all the features mentioned above is still unclear. Note however, that if accepted by the HPF forum, `ON` directives, `RESIDENT` directives, and `TASKING` regions, will be included only in the Approved Extensions of HPF-2.

References

[1] 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.

[2] 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.

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

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

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

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

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

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

[9] 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

A.1 Combined Directives

combined-directive	**is**	*combined-attribute-list* :: *entity-decl-list*
combined-attribute	**is**	ALIGN *align-attribute-stuff*
	or	DISTRIBUTE *dist-attribute-stuff*
	or	DYNAMIC
	or	INHERIT
	or	TEMPLATE
	or	PROCESSORS
	or	DIMENSION (*explicit-shape-spec-list*)
	or	*range-attribute*

A.2 PROCESSORS Directive

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

Constraint: *order-expr* and *shape-expr* must be restricted expressions.

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

A.3 DISTRIBUTE and REDISTRIBUTE Directives

distribute-directive	**is**	DISTRIBUTE *distributee dist-directive-stuff*
redistribute-directive	**is**	REDISTRIBUTE *distributee dist-directive-stuff*
	or	REDISTRIBUTE *dist-attribute-stuff* :: *distributee-list*
dist-directive-stuff	**is**	*dist-format-clause* [*dist-onto-clause*]
	or	*generalized-dist-format-clause* [*dist-onto-clause*]
dist-attribute-stuff	**is**	*dist-directive-stuff*
	or	*dist-onto-clause*

distributee	**is**	*object-name*
	or	*template-name*
	or	*structure-component*
dist-format-clause	**is**	`(` *dist-format-list* `)`
dist-format	**is**	`BLOCK` [`(` *int-expr* `)`]
	or	`CYCLIC` [`(` *int-expr* `)`]
	or	`GEN_BLOCK (` *int-array-expr* `)`
	or	`INDIRECT (` *int-array-expr* `)`
	or	`*`
generalized-dist-format-clause	**is**	`(` *generalized-dist-format* `)`
generalized-dist-format	**is**	`INDIRECT (` *int-array-expr* `)`
int-array-expr	**is**	*int-expr*

dist-onto-clause	**is**	`ONTO` *dist-target*
dist-target	**is**	*processors-section*
	or	`RESHAPE(`*processors-section*`,` *shape* [`,` *order*]`)`
processors-section	**is**	*processors-name* [`(`*section-subscript-list*`)`]
shape	**is**	[`SHAPE =`] *shape-expr*
shape-expr	**is**	*int-expr*
order	**is**	[`ORDER =`] *order-expr*
order-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: If a *dist-format-list* is specified, its length must equal the rank of each *distributee*.

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

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

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

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

Constraint: If *order-expr* and *shape-expr* of a *dist-target* appear in a `DISTRIBUTE` directive, they must be restricted expressions.

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

Constraint: Any *int-array-expr* apperaing 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 restricted expression.

A.4 RANGE Directive and Attribute

range-directive **is** `RANGE` *object-name range-mapping-list*

range-attribute **is** `RANGE` *range-mapping-list*

range-mapping **is** `(` *range-dist-format-list* `)` [*dist-onto-clause*]
or *dist-onto-clause*
or *align-attribute-stuff*

range-dist-format **is** `BLOCK` [`(` [*int-expr*] `)`]
or `GEN_BLOCK` [`(` *int-array-expr* `)`]
or `CYCLIC` [`(` [*int-expr*] `)`]
or `INDIRECT` [`(` *int-array-expr* `)`]
or `*`
or `ALL`

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

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

Constraint: Any *int-expr* appearing in a *range-dist-format* of a dummy object that has the `INHERIT` attribute must be a specification expression.

Constraint: Any *int-array-expr* appearing in a *range-dist-format* of a dummy object that has the `INHERIT` attribute must be a restricted expression.

A.5 INDEPENDENT Directive

independent-directive	**is**	`INDEPENDENT` [*independent-clause-list*]
independent-clause	**is**	*new-clause*
	or	*reduction-clause*
	or	*work-distribution-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*
work-distribution-clause	**is**	*on-clause*
	or	*distribute-clause*
on-clause	**is**	`ON HOME` (*array-element*)
	or	`ON` (*processor-array-element*)
distribute-clause	**is**	`DISTRIBUTE` (*dist-format*) [*dist-onto-clause*]

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

Constraint: If one of the *independent-clause*s is present, then the directive must apply to a `DO` loop.

Constraint: A *variable* named in the `NEW` option or any component or element thereof must not:

- Be a pointer or dummy argument; nor
- Have the `SAVE` or `TARGET` attribute.

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

Constraint: A *processors-section* appearing in an *dist-onto-clause* of a *distribute-clause* must have rank 1.

A.6 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 two *reduction-var-ref*s in a reduction statement 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.7 INHERIT Directive

inherit-directive **is** `INHERIT` [, *range-attribute*] :: *object-name-list*

range-attribute **is** `RANGE` *object-name range-mapping-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.