A User's Guide for HPF+

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

Deliverable: D2.1d
Date: March 1998
Status: draft
Confidentiality: public
Responsible: University of Vienna

Revised: April 7, 1998

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

Contents

1 Introduction

This User's Guide outlines the use of HPF+ and the Vienna HPF+ Compiler (VFC 1.1) for the parallelization of sequential Fortran programs.

VFC is a command-line compiler, that performs an automatic source-to-source translation of Fortran 90/HPF/HPF+ programs to Fortran 90/MPI message passing programs. The generated message passing programs are compiled with the Fortran 90 compiler of the parallel target computer and linked with the VFC runtime library and the MPI library in order to produce an executable parallel program.

Currently VFC is available on three different distributed memory platforms: the QSW CS-2, the NEC Cenju-3 and on clusters of workstations. VFC can be ported to any parallel computer that provides a Fortran 90 compiler and an MPI implementation.

VFC is based on full Fortran 90 and supports most of the HPF+ and HPF-2 language features. Besides the basic block and cyclic data distributions, VFC provides new data distribution formats required for irregular codes such as the general block distributions and indirect distributions, and it fully implements dynamic data redistribution.

Moreover, VFC implements a number of novel features that have been developed within the HPF+ projects including *communication schedule reuse* based on language support, and *halo areas* that may be used to describe at a high level the, possibly irregular, regions of non-local data accesses to distributed arrays.

The structure of this User's Guide is as follows:

The remainder of Section 1 describes the sources of parallelism exploited by VFC, and outlines the parallelization strategies adopted by VFC.

Section 2 gives an overview of the Fortran 90 and HPF+/HPF language features supported by VFC.

Section 3 describes the use of the HPF+ data distribution features offered by VFC.

Section 4 describes the language features of HPF+ provided for the parallelization of `DO` loops.

Section 5 describes the use of extrinsic procedures to exploit coarse grain parallelism.

Section 6 describes the command-line syntax of VFC and the process of compiling and executing HPF+ programs on different parallel architectures.

Section 7 lists the current restrictions of the VFC compiler.

Appendix A specifies the syntax of the HPF+ language features as currently available in VFC.

1.1 Programming with HPF+

Using a data-parallel language like HPF+, the user writes a sequential program and specifies at a high level how the data space of the program should be distributed. The data distribution is specified by means of compiler directives added to the declaration of arrays. Based on the user specified data distribution the compiler automatically derives an explicitly parallel SPMD program, i.e., it decides how to partition the control, sets up the necessary communication and generates the target code.

This style of programming hides the machine specific details inside the compiler and enables the user to concentrate on the algorithmic issues. As a consequence, programs written in this way are not only much simpler than message passing programs but also more flexible. Changing the data layout requires only minor program changes. Furthermore, it eases the migration of existing sequential programs to distributed memory machines and the porting of codes between various different architectures.

The provision of extrinsic procedures improves the flexibility of HPF+ by allowing the user to easily switch between global and local programming models.

A major characteristic of this style of programming is that the performance of the resulting code depends to a very large extent on the data distribution selected by the programmer. The data distribution usually determines not only where computation will take place. It is also the main factor in deciding what communication is necessary.

1.2 Sources of Parallelism

VFC exploits the parallelism inherent in the following program constructs:

- Array statements
  Fortran 90 array assignment statements are treated as parallel constructs. They are transformed by VFC such that each processor executes its part of the computation, according to the distribution of the left-hand-side array.

- `INDEPENDENT` loops
  The `INDEPENDENT` directive enables the user to assert that the iterations of a `DO` loop are independent and can be executed in parallel. The distribution of loop iterations to processors may be specified by the user, and a facility for reusing the communication schedules of such loops is provided. Parallelization of `INDEPENDENT` loops is implemented based on runtime parallelization techniques.

- Reduction operations
  Within independent loops reduction operations such as for example summing all elements of a vector, computing the dot-product of two vectors, computing the minimum or maximum value of a vector, and others are supported.

The VFC compiler may be used to exploit *fine grain parallelism* as well as *coarse grain parallelism*. Fine grain parallelism is exploited by parallelizing array assignments and `DO` loops. Coarse grain parallelism may be exploited by means of local extrinsic procedures. This is described in more detail in Section 5.

1.3 SPMD Programming Model

The parallelization strategy of VFC is based on the SPMD programming model. VFC translates a source program into an SPMD message passing target program, which is parameterized in such a way that it can be executed on an arbitrary number of processors[1].

- First, the data arrays of the program are partitioned according to the data distribution directives inserted by the user into pieces each of which is assigned to a processor which becomes the *owner* of the data assigned to it.
- Second, computations are distributed according to the specified data distribution usually by applying the *owner computes rule*. The owner computes rule implies that each processor only executes assignments to data elements owned by it. In the context of do-loops work distribution may be specified by the user and thus is not restricted to the owner computes rule.
- Finally, access to non-local data is implemented by *message passing*, i.e. by inserting appropriate communication statements to send and receive data between processors.

At runtime each processor of the target machine executes the same target program, but operates only on its local portion of the data. The generated SPMD program utilizes calls to the MPI library to implement communication between the processors of a parallel machine.

1.4 Parallelization Strategies

Several different parallelization techniques have been implemented, however, the main focus of the implementation has been on a generalization of the inspector/executor parallelization strategy for irregular independent loops which are currently not adequately supported by commercial HPF compilers.

Loops with irregular runtime dependent access patterns are essential for advanced industrial applications and in particular for the benchmark programs developed within the HPF+ project. The inspector/executor strategy as implemented in VFC can handle arbitrary, possibly runtime defined, multi-dimensional distributions and supports the important concept of communication schedule reuse, which has been developed within the HPF+ project. Depending on the structure of independent loops and the properties of the accessed arrays several different variants of the loop parallelization strategy are applied.

A specialized strategy has been implemented for the efficient parallelization of regular array assignment statements, for which the application of the inspector/executor strategy would lead to unacceptable runtime overheads.

A default parallelization strategy based on single-element send/receive or broadcast communication, is applied if neither of the above strategies is applicable. If this strategy has to be adopted for a loop the user is informed about the potential overheads by means of *diagnostic messages*.

A detailed description of the various parallelization strategies of VFC can be found in [7].

[1] This is only true if the program has been written in such a way that it is not dependent on a specific number of processors.

2 Input Language

Input languages for VFC are full Fortran 90 with a subset of HPF+ and HPF-2 [10] directives including most of the language features specified in the HPF-2 Approved Extensions.

The complete syntax of the HPF+ language features as supported by VFC can be found in Appendix A.

2.1 HPF+ and HPF Language Features

Currently VFC offers the following HPF+ language features:

- *Arbitrary processor arrays*
- *Regular block and cyclic distributions*
- *General block and indirect distributions*
- *Arbitrary multi-dimensional distributions*
- *Alignments*
- *Halos*
- *Inheritance of distributions*
- *Distribution of* `ALLOCATABLE` *and* `POINTER` *objects*
- *Dynamic distributions and redistribution*
- *Distribution ranges*
- *Independent Loops*
- *Extrinsic Procedures*

2.1.1 Processors Statements

In order to declare processor arrays VFC provides the following features:

- the `PROCESSORS` directive, and
- the `NUMBER_OF_PROCESSORS()` intrinsic function.

VFC requires that an input program contains at least one processor array declaration. The intrinsic function `NUMBER_OF_PROCESSORS()` may be used to specify processor arrays whose sizes are determined upon program start.

2.1.2 Distribute Directives

For arrays of type `INTEGER, REAL`, or `DOUBLE PRECISION` arbitrary, multi-dimensional distributions using any combination of the following distribution formats may be specified by means of the `DISTRIBUTE` directive.

- `BLOCK`, `BLOCK(M)`
- `CYCLIC`, `CYCLIC(M)`
- `GEN_BLOCK`
- `INDIRECT`
- `*` (for non-distributed dimensions)

2.1.3 Alignment Directives

VFC supports simplified forms of the *align directive* supporting either perfect alignment of conformable arrays or the alignment of individual dimensions of an array.

2.1.4 Halos

The set of non-local elements of a distributed array that may be accessed during the execution of the program may be specified by defining *halo regions* by means of the `HALO` directive or attribute. The `UPDATE_HALO` directive is provided in order to update halo regions.

2.1.5 Dynamic Data Distribution

VFC supports dynamic redistribution in the full generality as specified by HPF+. The following language features for dynamic data distribution are provided.

- `DYNAMIC` attribute and directive
- `REDISTRIBUTE` statement

For non-formal arrays the `DYNAMIC` attribute may be combined with the `DISTRIBUTE` directive in order to provide an *initial distribution*. The `REDISTRIBUTE` statement supports the same distribution formats as outlined in 2.1.2.

2.1.6 Distribution of Dummy Arrays

The distribution of dummy arrays may specified in one of the following ways:

1. With the `INHERIT` attribute.
2. With the `DYNAMIC` attribute. In this case the corresponding actual array must also have the `DYNAMIC` attribute. This feature may be used to define or modify the distribution of an array as the result of a procedure call.
3. With the `DISTRIBUTE` directive. In this case, VFC requires that the distribution of the corresponding actual array must not be dynamic and must always match the distribution of the dummy array.

2.1.7 Independent Do Loops

The `INDEPENDENT` directive may be used in order to indicate to the system that a `DO` loop does not contain loop carried dependencies, and thus the loop iterations may be executed in any order.

The following clauses may be specified within the `INDEPENDENT` directive.

- the `ON HOME` clause, to specify the distribution of loop iterations,
- the `REDUCTION` clause, for reduction operations,
- the `NEW` clause, to declare *private* variables,
- the `REUSE` clause with or without a logical condition, for *schedule reuse*,
- the `RESIDENT` clause to avoid communication.

2.1.8 The `PURE` Attribute and the `PUREST` Directive

The `PURE` attribute is recognized. However, the constraints imposed by it are not yet checked by VFC. The `PUREST` directive may be specified for procedures that exclusively operate on local data.

2.1.9 Extrinsic Procedures

The `EXTRINSIC(HPF_LOCAL)` and `EXTRINSIC(HPF_SERIAL)` prefix for procedures and the associated argument transfer mechanisms are supported by VFC.

2.2 Fortran 90 Language Features

VFC supports full Fortran 90. As long as a certain language construct does not involve distributed objects there are no restrictions. In the context of distributed objects currently some restrictions have to be obeyed. These restrictions, which are all checked by the compiler, are summarized in the Appendix. In the following we list some important Fortran 90 features provided by VFC. For a detailed specification of Fortran 90 language features the reader is referred to [8]. The current restrictions of VFC are described in Section 7.

2.2.1 Dynamic Memory Allocation

VFC fully supports all mechanisms of Fortran 90 for dynamic memory allocation. An array that should be allocated dynamically must have the `ALLOCATABLE` or `POINTER` attribute. Memory allocation can either be done either directly using the `ALLOCATE` statement, or it can be hidden inside a procedure, in which case both the actual and formal argument must have the `POINTER` attribute. Moreover, automatic objects are fully supported. Dynamic memory allocation is available for both distributed and non-distributed objects.

2.2.2 Modularization and Separate Compilation

VFC supports the modularization concepts of Fortran 90. In particular, internal procedures, external procedures, interface blocks and modules are supported. Programs may be parallelized by compiling different program units separately.

2.2.3 Argument Transfer Mechanisms

VFC supports all Fortran 90 argument transfer mechanisms, including explicit shape arrays, assumed shape arrays and assumed size arrays. Whereas these mechanisms can be used in the full generality as provided by Fortran 90 for non-distributed arrays, there are some restrictions for the transfer of distributed arguments at procedure boundaries. In particular, VFC does not support reshaping of distributed arrays and the transfer of sub-arrays.

The assumed shape argument transfer mechanism, which has been the main focus of the implementation, should be used, whenever possible for distributed arguments. In this context, the intrinsic functions `LBOUND, UBOUND, SHAPE`, and `SIZE` may be used to inquire about properties of the incoming argument.

2.2.4 Input/Output

For the I/O of non-distributed objects VFC supports all the input/output mechanisms of Fortran 90, with some restrictions for the I/O of distributed data objects. Since input/output is not the major objective of this project, only basic functionality for the I/O of distributed objects is currently provided.

This includes reading/writing of either distributed whole arrays or elements thereof. Whereas both formatted and unformatted input/output is supported, no formats other than "*" can currently be used for distributed arrays.

Note however, that restricted forms of *parallel I/O* may be achieved based on local extrinsic procedures.

2.2.5 Summary of Fortran 90 Features

- *Fixed and Free Source Form*
- *Dynamic Memory Allocation*
- *Automatic Arrays*
- *Assumed Shape Arrays*
- *Array Statements*
- *Array Constants*
- *Array Constructors*
- *Derived Types*
- *Block* `DO` *Construct*
- `DO WHILE` Construct
- `SELECT` *Statement*
- *Internal Procedures*
- *Modules*
- *Overloading*
- *Interface Blocks*
- *Keyword and Optional Arguments*

3 Data Distribution

The distribution of a data array is usually achieved by distributing each of its dimensions using either the BLOCK, CYCLIC, GEN_BLOCK, or INDIRECT distribution formats, or "*" for non-distributed dimensions. Another way to specify the distribution of an array is to align it with another already distributed array. VFC supports arbitrary multi-dimensional distributions that may be specified either by means of the DISTRIBUTE directive or by means of the ALIGN directive.

In addition to the distribution of an array, the set of non-local elements accessed on each processor may be specified by means of the *halo attribute* as described in Section 3.9.

VFC distinguishes two types of distributed arrays: *statically distributed arrays* and *dynamically distributed* arrays. Whereas statically distributed arrays have a fixed distribution within their scope of declaration, the distribution of dynamically distributed arrays may be changed at runtime. Dynamically distributed arrays must be declared with the DYNAMIC attribute and are described in Section 3.6.

In the following we first define some terms that are used in the context of data distribution and then describe the use of the main HPF+ language features by means of examples. For a detailed specification of HPF+ the reader is referred to the HPF+ Language Specification [6].

3.1 Terminology

- A **distributed array** is an array that has either of the following attributes:
  - the DISTRIBUTE attribute,
  - the ALIGN attribute,
  - the DYNAMIC attribute, or
  - the INHERIT attribute.

- A **direct distribution** is a distribution that is specified by means of the DISTRIBUTE attribute or directive or by means of the REDISTRIBUTE directive.

- An **explicitly distributed** array is an array that has the DISTRIBUTE, the ALIGN, or the DYNAMIC attribute.

- A **distributee** is an array that has the DISTRIBUTE attribute or that appears as the object of a REDISTRIBUTE statement.

- An **alignee** is an array that has the ALIGN attribute.

- A **statically distributed** array is a distributed array that does not have the DYNAMIC attribute.

- A **dynamically distributed** array is a distributed array that has the DYNAMIC attribute.

Example 3.1: Processor Arrays

```
PROGRAM P

!HPF$ PROCESSORS R(NUMBER_OF_PROCESSORS())
!HPF$ PROCESSORS R2(NUMBER_OF_PROCESSORS()/2,2) = RESHAPE(R)
      ...
CONTAINS
    SUBROUTINE PSUB ( ... )

       ! processor arrays R and R2 are accessible here (host association)

    END SUBROUTINE PSUB

END PROGRAM P
```

This example shows the declaration of two processor arrays. The first processors directive declares a one-dimensional processor array R *whose size is determined upon program start by evaluating the intrinsic function* NUMBER_OF_PROCESSORS()*. The second processors directive declares a processor array* R2 *which corresponds to the same processors as* R*. Correspondence between* R *and* R2 *is established by means of the Fortran array element order. Both processor arrays are accessible in the internal procedure* PSUB *by means of host association. The number of processors the program can be executed on must in this case be at least 2 and should be an even number.*

Note that in HPF the user has no means to establish a relationship between the processor arrays R *and* R2*.*

3.2 Processors Arrays

Processor arrays may be declared by the user to define the set of *abstract processors* used to execute a program. Processor arrays are introduced by means of the *processors directive* which must occur in the specification part of a program unit. Arbitrary multi-dimensional processor arrays may be defined and used in a program. Note that VFC requires that at least one processor array is declared in an HPF+ program.

Unlike HPF, in HPF+ an explicit relationship between different processor arrays may be established by using the RESHAPE intrinsic function as shown in Example 3.1. This alleviates the distribution of data arrays of different rank to the same set of processors. Processor arrays that have the same shape are always associated with the same set of processors.

HPF+ programs may be parallelized for an arbitrary number of processors by using the intrinsic function NUMBER_OF_PROCESSORS() in processor declarations. In this case, the size of a processor array is determined upon program start.

Processor arrays are not first class objects and therefore cannot be passed as arguments to procedures. Globally accessible processor arrays must be declared in a module which has to be *used* by all program units that need to access the global processor arrays. This is shown in Example 3.2.

Example 3.2: Globally Accessible Processor Arrays

```
MODULE M                                                  ! module

!HPF$ PROCESSORS R_GLOBAL(NUMBER_OF_PROCESSORS())

CONTAINS

    SUBROUTINE MSUB                                       ! module subprogram

    ! processor array R_GLOBAL is accessible here (host association)

    END SUBROUTINE MSUB

END MODULE M


PROGRAM P2                                                ! main program
   USE M

   ! processor array R_GLOBAL is accessible here (use association)

CONTAINS

    SUBROUTINE P2SUB                                      ! internal subprogram

    ! processor array R_GLOBAL is accessible here (host association)

    END SUBROUTINE P2SUB

END PROGRAM P2


SUBROUTINE SUB                                            ! external subprogram

   USE M

   ! processor array R_GLOBAL is accessible here (use association)

END SUBROUTINE SUB
```

This example shows how a processor array may be made globally accessible by using modules. The processor array `R_GLOBAL` *is declared in the module* `M` *is accessible in all module subprograms of* `M`*, for example in* `MSUB`*, by means of host association. Moreover, it is accessible in any program unit that uses module* `M` *by means of use association.*

3.3 Direct Data Distribution

The distribution of arrays may be specified by means of the DISTRIBUTE directive by specifying for each dimension of the array a distribution format. In this case we speak of a *direct distribution* opposed to a distribution defined by means of alignment.

A DISTRIBUTE directive has the following form:

!HPF$ DISTRIBUTE (*dist-format-list*) [ONTO *processor-array*] :: *array*

A DISTRIBUTE directive is subject to the following rules:

1. All expressions appearing as arguments of a distribution format must be *specification expressions* which must not reference distributed objects.
2. The number of *dist-formats* must be equal to the rank of the array being distributed.
3. The number of *dist-formats* that are not "*", referred to as *distribution rank*, must always match the rank of the processor array mentioned in the ONTO clause.
4. If no ONTO clause is specified, then a processor array whose rank is equal to the distribution rank of the array being distributed must be accessible in the corresponding program unit. This processor array is then taken by VFC as the target of the distribution.

Example 3.3 shows static distributions defined by means of the DISTRIBUTE directive. Section 3.9 describes how *halos* may be specified for distributed arrays.

Example 3.3: Static Distributions

```
      INTEGER A2(100,100), B2(100,100), A3(100,100,100)
!HPF$ PROCESSORS R(4)
!HPF$ PROCESSORS R2(2,2) = RESHAPE(R)

!     multi-dimensional distributions
!
!HPF$ DISTRIBUTE(BLOCK,*,CYCLIC) ONTO R2 :: A3

!     general block distribution
!
      INTEGER, PARAMETER :: BS(4) = (/ 30, 20, 20, 30 /)  ! block-sizes
!HPF$ DISTRIBUTE(GEN_BLOCK(BS), *) ONTO R1 :: A2

!     indirect distributions
!
      INTEGER MAP(100)                                      ! mapping array
!HPF$ DISTRIBUTE(INDIRECT(MAP),*) :: B2
```

This example shows statically distributed arrays. Note that all arguments of distribution formats of statically distributed array must be specification expressions. Thus, for example, MAP *which is used in the indirect distribution must be an array constant or a dummy argument. Note also that* B2 *will be distributed onto processor array* R*, which is chosen automatically by VFC.*

3.4 Runtime-defined Static Distributions

VFC supports run-time defined static distributions, by allowing to use arguments, which are defined at runtime, in the distribution formats of statically distributed arrays. For example, this feature should be used if block-sizes of BLOCK, CYCLIC or GEN_BLOCK distribution formats or mapping arrays used in INDIRECT distributions have to be computed at runtime, but the corresponding arrays have a fixed distribution.

Since HPF requires all expression used in distribution formats to be specification expressions, arrays that should be distributed in such a way must either be local arrays of a procedure or dummy arguments. VFC currently is even more restrictive and requires that arrays distributed in such a way are local arrays of a procedure as shown in Example 3.4.

Example 3.4: Runtime-defined Static Distributions

```
PROGRAM P

      INTEGER, DIMENSION(1000) :: MAP
      ...

      READ(*,*) MAP

      CALL MAIN()
      ...

CONTAINS

      SUBROUTINE MAIN()

         REAL, DIMENSION(6,1000) :: A

!HPF$    PROCESSORS R(NUMBER_OF_PROCESSORS())
!HPF$    DISTRIBUTE(*,INDIRECT(MAP)) ONTO R :: A

         ...

      END SUBROUTINE MAIN

END PROGRAM P
```

This example shows how an array with a static distribution, which is dependent on runtime-data, may be used. Since all expressions used in distribution formats must be specification expressions, array A *has to be a local array, for example, of an internal procedure.*

Another way to specify a distribution which is dependent on data computed at runtime is by means of DYNAMIC and REDISTRIBUTE as described in Section 3.6. Note, however, that currently the use of arrays with the DYNAMIC attribute may result in a severe performance loss[2] due to the Fortran 90 compiler used to compile the SPMD codes generated by VFC.

[2] On the CS-2 we measured a performance decrease of about a factor of 2 with the SUN F90 compiler.

3.5 Distribution of Dynamically Allocated Arrays

The Fortran mechanisms for dynamic memory allocation are supported for distributed objects without any restrictions.

An array that should be allocated dynamically must have the `ALLOCATABLE` or the `POINTER` attribute. Allocation and deallocation can be done with the `ALLOCATE` and `DEALLOCATE` statement, respectively. Memory for an object with the `POINTER` attribute can also be allocated inside of a procedure, in which case both the actual and formal argument must have the `POINTER` attribute[3].

To check whether a pointer object is already allocated the Fortran intrinsic function `ASSOCIATED` may be used.

Example 3.5: Dynamic Allocation of Distributed Arrays

```
PROGRAM P

REAL, DIMENSION(:), ALLOCATABLE :: A
REAL, DIMENSION(:), POINTER     :: P

!HPF$ PROCESSORS R(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE(BLOCK) :: A, P
...                                        ! interface for SUB required

ALLOCATE(A(N), STAT=...)

CALL SUB(P)
...

END PROGRAM P

SUBROUTINE SUB(Q)

      REAL, DIMENSION(:), POINTER :: Q
!HPF$ INHERIT                     :: Q
         ...
         IF (.NOT.ASSOCIATED(Q)) THEN
            ALLOCATE(Q(somesize), STAT=...)
            ...
         END IF
         ...
END SUBROUTINE SUB
```

This example shows how memory for a distributed array can be allocated dynamically using either the `ALLOCATABLE` *or* `POINTER` *attribute. Array* `A` *is directly allocated by means of an* `ALLOCATE` *statement in the scope of its declaration. Array* `P` *which has the* `POINTER` *attribute is passed to a procedure where the corresponding dummy argument, which also must have the* `POINTER` *attribute, is allocated and, due to the* `INHERIT` *attribute, distributed according to the distribution specified for the corresponding actual argument* `P`*. Note that an explicit interface is required by Fortran for* `SUB` *since it has a dummy argument with the* `POINTER` *attribute.*

[3]Note that in Fortran 95 a dummy argument must not have the `ALLOCATABLE` attribute. This severe drawback will be hopefully removed in Fortran 2000.

3.6 Dynamic Data Distribution and Redistribution

The distribution of arrays that have the `DYNAMIC` attribute may change at runtime either by executing a `REDISTRIBUTE` statement or as the result of a procedure call. An *initial distribution* may be specified for arrays with the `DYNAMIC` attribute by means of the `DISTRIBUTE` attribute.

A `REDISTRIBUTE` directive is an executable directive which has either of the following forms[4]:

`!HPF$ REDISTRIBUTE (` *dist-format-list* `) [ ONTO` *processor-array* `] [` *halo* `] ::` *array*

`!HPF$ REDISTRIBUTE array (` *dist-format-list* `) [ ONTO` *processor-array* `] [` *halo* `]`

Note that the expressions appearing as arguments to *dist-formats* in a `REDISTRIBUTE` statement are not restricted to specification expressions.

The following rules have to be obeyed when using dynamically distributed arrays:

1. Only an array that is a *distributee* and that has the `DYNAMIC` attribute may be the object of a `REDISTRIBUTE` statement.
2. If no initial distribution has been specified, a dynamically distributed array may not be accessed (or allocated) unless it has been distributed.
3. If an array with the `DYNAMIC` attribute is an alignee the corresponding align-target must also have the `DYNAMIC` attribute.
4. If a dynamically allocated array has the `DYNAMIC` attribute, then it must have been associated with a distribution at the time it is allocated.

Data redistribution is supported by VFC without any restriction for multi-dimensional arrays with up to seven dimensions. Note however that redistribution of arrays is usually a time-consuming task and thus should be used carefully. Another reason why dynamic data redistribution must be used carefully is that the use of the `DYNAMIC` attribute may inhibit certain compile-time optimizations since it may be impossible to determine for a certain program construct that references dynamically distributed arrays which distributions may *reach* this construct. To provide the compiler with more information about the distributions a dynamically distributed array may have, the `RANGE` directive or attribute should be used, as described in the next section.

Note:
Currently the VFC compiler transforms an array with a `DYNAMIC` attribute to a *deferred shape array* which has the `POINTER` attribute. This is needed since such arrays are redistributed by calling corresponding redistribution routines from the VFC runtime system. Within these routines such arrays may have to be reallocated during redistribution. This requires the use of `POINTERS` since Fortran 90/95 does not allow dummy arrays to have the `ALLOCATABLE` attribute. This restriction will be removed in Fortran 2000.

[4] The specification of halos within a `REDISTRIBUTE` directive is described in Section 3.9.

Example 3.6: Dynamically Distributed Arrays

```
REAL, DIMENSION(N)              :: A, B
REAL, DIMENSION(:), ALLOCATABLE  :: C
REAL, DIMENSION(:), POINTER      :: D

!HPF$ PROCESSORS R(NUMBER_OF_PROCESSORS())
!HPF$ DYNAMIC                   :: A, B, C, D
!HPF$ DISTRIBUTE(BLOCK) ONTO R :: B

      A = ...  ! not allowed - A not yet distributed
      B = ...  ! ok - B has an initial distribution
      C = ...  ! not allowed - C neither distributed nor allocated
      D = ...  ! not allowed - D neither distributed nor allocated

!HPF$ REDISTRIBUTE(CYCLIC) ONTO R :: A, B    ! s1
      A = ...  ! ok

!HPF$ REDISTRIBUTE((BLOCK)) ONTO R :: C      ! s2
      ALLOCATE(C(10000))                     ! s3

      CALL SUB(C, D)                         ! s4
               ! C is cyclically distributed after call to SUB
      D = ...  ! D is now allocated and has a BLOCK distribution

      ...

   SUBROUTINE SUB(X, Y)
      REAL, DIMENSION(:)          :: X
      REAL, DIMENSION(:), POINTER :: Y
!HPF$ DYNAMIC :: X, Y
      ...

!HPF$ REDISTRIBUTE(CYCLIC) ONTO R :: X
!HPF$ REDISTRIBUTE(BLOCK)  ONTO R :: Y
      ALLOCATE(Y(10000))
      ...
    END SUBROUTINE SUB
```

This example shows different ways to distribute arrays dynamically. Array `A` *is associated with a* `CYCLIC` *distribution by executing the first* `REDISTRIBUTE` *statement (s1).* `B` *has an initial* `BLOCK` *distribution which is changed to* `CYCLIC` *after execution of statement s1.* `C` *is associated with a* `BLOCK` *distribution in the second* `REDISTRIBUTE` *statement (s2) and may be accessed after its allocation in statement s3. Note that changing the order of s2 and s3 would cause the program to terminate, since an array with the* `DYNAMIC` *attribute must not be allocated before it has been associated with a distribution. Array* `D` *is distributed and allocated within procedure* `SUB`*. The distribution of* `C` *is changed from* `BLOCK` *to* `CYCLIC` *as the result of the call in s4.*

3.7 Distribution Ranges

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

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

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 3.7 illustrates the use of the `RANGE` attribute.

The `RANGE` attribute should be specified whenever possible to avoid potential overheads due to insufficient information about the distributions an array may be associated with. In particular, it is crucial for the compiler to "know" whether a dimension of an array is replicated or distributed (regardless of type of the distribution).

The `RANGE` attribute may be also specified for dummy arguments with an *inherited distribution* as described in Section 2.1.6.

Example 3.7: The `RANGE` Attribute

```
!HPF$ PROCESSORS R(NUMBER_OF_PROCESSORS())

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

!HPF$ DYNAMIC :: A, B, C

!HPF$ RANGE (*,BLOCK), (*, CYCLIC)  :: A
!HPF$ RANGE (BLOCK,*)               :: B
!HPF$ RANGE (ALL, *)                :: C
      ...

      REDISTRIBUTE(BLOCK,*) :: A    ! illegal: distribution not specified in range
```

The first range directive constitutes an assertion to the compiler that the first dimension of `A` *is never distributed and that the second dimension of* `A` *has either a block or a cyclic distribution. The range directive of* `B` *asserts that* `B` *will always be distributed in the same way. The last range directive asserts that the second dimension of* `C` *is always replicated while the first dimension may have any distribution.*

The redistribution statement for A is illegal since A is associated with a distribution that is not contained in the range of permissible distributions. Note that this is not checked by VFC, and therefore the corresponding program is non HPF+ conforming and likely to fail or to produce incorrect results.

3.8 Alignment

In HPF+ alignment may be used to specify that an array or an array dimension is distributed in the same way as another array or array dimension. An alignment relationship between two arrays is fixed and cannot be destroyed at runtime. Alignment should be used to specify groups of arrays that are always distributed in the same way, particularly in the context of dynamic data distribution.

HPF+ align directives have one of the following forms:

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

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

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

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

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

The other forms with an *align-source-list* and an *align-subscript-list* are provided to specify alignment with respect to individual array dimensions. Each *align-source* and *align-subscript* corresponds to one dimension of the alignee or align-target, respectively, and is either an *align-dummy* variable or a "*". Each dimension of the alignee for which an *align-dummy* variable is specified is aligned with the dimension of the align-target that references the same *align-dummy* variable in the *align-subscript-list*.

The alignment mechanisms of HPF+ are a subset of those provided by HPF supporting perfect alignments, either of whole arrays or of individual array dimensions, collapsing of individual dimensions of the *alignee*, transposed alignments, and replication over individual dimensions of the *align-target*.

Alignment strides and offset or embedding smaller dimensions into larger dimensions, as provided by HPF, are not supported. Alignments as currently provided by VFC are shown in Example 3.8.

Alignments as supported by VFC are subject to the following rules:

1. An array may only be aligned with an *align-target* that is a *distributee*.
2. Perfect alignment may only be specified between conformable arrays.
3. An array dimension may only be aligned with a dimension of another array that has the same size.
4. A statically allocated array must not be aligned with a dynamically allocated array.
5. The alignee and align-target must either have both the `DYNAMIC` attribute or neither may have.
6. Permuted alignments are not yet supported by VFC.

Example 3.8: HPF+ Alignment Directives

```
REAL, DIMENSION(N)   :: A1, B1
REAL, DIMENSION(N,M) :: A2, B2
REAL, DIMENSION(0:N) :: X

! perfect alignment
!HPF$ ALIGN A2 WITH B2

! alignment of individual array dimensions
!HPF$ ALIGN A2(I,J) WITH B2(I,J)
!HPF$ ALIGN A2(I,*) WITH B2(I,*)

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

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

! illegal alignment: X and B1 are not conformable
!HPF$ ALIGN X WITH B1

! permuted alignments not yet supported
!HPF$ ALIGN A2(I,J) WITH B2(J,I)
```

Note that the distribution of an array which is specified by means of alignment can always be expressed by means of a direct distribution. Alignment does not allow to specify distributions that could not be specified by means of DISTRIBUTE.

However, the use of alignments may provide the compiler with additional information in the context of dynamically allocated or dynamically distributed arrays. If two dynamically allocated arrays are aligned with each other, then this is an assertion to the compiler that the arrays will always have the same shape. Similarly, if two dynamically distributed arrays are aligned with each other, then this is an assertion to the compiler that the arrays have always the same distribution with respect to the aligned dimensions. This is shown in Example 3.9.

Example 3.9: Alignment of Dynamically Allocated and Dynamically Distributed Arrays

```
      REAL, ALLOCATABLE, DIMENSION(:) :: A, B
!HPF$ DYNAMIC                         :: A, B
!HPF$ ALIGN WITH B                    :: A

      ...

      A(l:u) = B(l:u)
```

This example shows the alignment of dynamically allocated and dynamically distributed arrays. The alignment directive enables the compiler to parallelize the array assignment without introducing communication since the align directive asserts that A *and* B *will always have the same distribution and the same shape.*

3.9 Halos

The set of non-local elements of a distributed array that may be accessed by a processor during the execution of a program unit may be specified by means of *halo regions*. A halo region is usually an irregular, non-contiguous area describing the non-local access patterns of an array on each processor. The information provided by halo regions can be utilized by the compiler to allocate memory for all non-local accesses, to establish a corresponding address translation scheme, and to compute the associated communication schedules. In particular, the use of halo regions may improve the parallelization of irregular independent loops significantly.

Halo regions are specified by means of the `HALO` attribute which may be combined with distribution specifications. The halo of a distributed array is specified by providing a *halo-specification* for each dimension of the array. A halo-specification must be a one-dimensional integer expression, usually a Fortran function, which, when evaluated on a particular processor, returns a one-dimensional integer array containing the global indices of non-local array elements accessed on that processor.

Halos may be used for statically or dynamically distributed arrays. In the first case the `HALO` attribute must be specified within the `DISTRIBUTE` attribute or by means of the `HALO` directive. In the latter case, for dynamically distributed arrays, the halo is specified within the `REDISTRIBUTE` directive.

Note:
If an array `B` is perfectly aligned[5] with another array `A` that has a halo, `B` implicitly takes over the halo of the align-target `A`.

Example 3.10: Using Halos (1)

```
!HPF$ PROCESSORS R(NUMBER_OF_PROCESSORS())

!HPF$ DISTRIBUTE(BLOCK), HALO(halo_function_1(I)) ON R(I) :: A
!HPF$ ALIGN WITH A :: B      ! B gets same halo as A
!HPF$ DYNAMIC        :: A, B

!HPF$ UPDATE_HALO :: A, B                                    ! update halo regions
      ...

      ! redistribute arrays with different distribution and a new halo
      !
!HPF$ REDISTRIBUTE(CYCLIC), HALO(halo_function_2(I) ON R(I) :: A
      ...
!HPF$ UPDATE_HALO :: A, B                                    ! update halo regions
```

In this example the `HALO` *attribute is specified together with the initial distribution of the dynamically distributed array* `A`*. Array* `B` *implicitly gets associated with the same halo as* `A` *since it is perfectly aligned with* `A`*. The* `UPDATE_HALO` *directive results in the update of the array elements in the halo of the executing processor by receiving these elements from the owner processors. At the time the* `REDISTRIBUTE` *directive for* `A` *is executed,* `A` *gets associated with a new halo according to the function* `halo_function_2`*. Array* `B` *is also redistributed and associated with the new halo since it is perfectly aligned with* `A`*.*

[5] This requires the use of the syntactic form of the `ALIGN` attribute or directive without an *align-source-list*.

Halos have been implemented in such a way that it is always the responsibility of the user to update the halo of distributed arrays by means of the UPDATE_HALO directive. The use of halos is shown in Example 3.10 and 3.11.

Example 3.11: Using Halos (2)

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

      TYPE HALO_TYPE
         INTEGER, POINTER :: INDEX
      END TYPE HALO_TYPE

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

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

      ALLOCATE(SLHALO(NUMBER_OF_PROCESSORS))

      READ(*,*) MAP                   ! mapping array

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

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

!HPF$ UPDATE_HALO :: A
```

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

3.10 Distribution of Dummy Arrays

The distribution of dummy arrays may be specified in one of the following ways:

1. With the INHERIT attribute.
2. With the DYNAMIC attribute. In this case the corresponding actual array must also have the DYNAMIC attribute. This feature may be used to define or modify the distribution of an array as the result of a procedure call.
3. With the DISTRIBUTE directive. In this case the distribution of the corresponding actual array must not be dynamic and must always match the distribution of the dummy array. (This is checked by the system.)

Note that dummy arrays for which a distribution has been specified in one of the above ways must always be *assumed shape arrays.* The Fortran 90 intrinsic functions `SIZE`, `LBOUND`, `UBOUND`, and `SHAPE` are fully supported for distributed arrays in order to enquiry about properties of an assumed shape dummy argument.

The interface of any procedure that contains distributed formal arguments must be explicit. This can be achieved by

- specifying an *interface block*,
- using *internal procedures*, or
- by means of modules.

If an interface of a procedure that has distributed dummy arguments is specified in an interface block, then the distribution directives and processor array declarations must be specified in the interface.

The distribution mechanisms of dummy arrays are subject to the following rules:

1. The interface of a procedure that has distributed dummy arguments must be explicit.
2. Dummy arrays for which a distribution has been specified in one of the above ways must always be *assumed shape arrays.* The transfer of sub-arrays is not yet supported.
3. If a dummy array has the `INHERIT` attribute it may be associated with an actual array with any distribution, or if the `RANGE` attribute is also present, with any distribution within the *range of permissible distributions.*
4. If a dummy argument has the `DYNAMIC` attribute, the actual array must also have the `DYNAMIC` attribute and may have any *permissible* distribution.
5. If a dummy array is specified with the `DISTRIBUTE` attribute, the distribution of the actual and formal argument must match. If this is not the case, a corresponding *restriction message* will be generated by VFC.
6. If a dummy array has the `DYNAMIC` attribute, an initial distribution must not be specified.
7. In cases where a distributed actual argument is associated with a non-distributed formal argument, an implicit redistribution to replicate the actual argument is generated by the VFC prior to the call, and after the call to restore the original distribution.

   If such a redistribution is generated the user is informed about this potential runtime overhead by corresponding *diagnostic messages.*

Note that rules 2,5,6 and 7 are restrictions imposed by the current version of VFC. All other rules are in accordance with the HPF+ and HPF Language Specifications.

The mechanisms of calling procedure with distributed arguments and the distribution of dummy arrays is shown in Example 3.12.

Example 3.12: Distribution of Dummy Arrays

```
PROGRAM P

      INTERFACE

         SUBROUTINE SUB1(X, Y)
            REAL, DIMENSION(:) :: X, Y
!HPF$       PROCESSORS Q(NUMBER_OF_PROCESSORS())
!HPF$       DISTRIBUTE(BLOCK) :: X
!HPF$       INHERIT           :: Y
         END SUBROUTINE SUB1

         SUBROUTINE SUB2(Z)
            REAL, DIMENSION(:) :: Z
         END SUBROUTINE SUB2

      END INTERFACE
      ...

!HPF$ DISTRIBUTE(BLOCK)  :: A, B
!HPF$ DISTRIBUTE(CYCLIC) :: C

      ...
      CALL SUB1(A, B)     ! s1   -  no redistribution required

      CALL SUB1(A, C)     ! s2   -  no redistribution required

      CALL SUB2(A)        ! s3   -  implicit redistribution generated by VFC

      CALL SUB1(C, A)     ! s4   -  not yet supported
      ...
END PROGRAM P
```

This example shows the data distribution mechanisms for procedure arguments. The interface block for subroutine `SUB1` *is required by HPF+ since* `SUB1` *has distributed dummy arguments, and by Fortran for* `SUB1` *and* `SUB2` *since both subroutines have assumed shape dummy arrays. Note that the distribution directives of dummy arrays have to be included in an interface block, as shown for* `SUB1`*.*

The call to `SUB1` *in statement* `s1` *will not require any data redistribution since the distribution of the first actual argument* `A` *matches the distribution of the dummy argument* `X`*, and the second dummy argument* `Y` *has the* `INHERIT` *attribute and thus can be called with an actual argument of any distribution.*

The call to `SUB1` *in statement* `s4` *will result in a restriction message, since VFC does not yet support associating an explicitly distributed dummy array with an actual array of a different distribution.*

The call to `SUB2` *in statement* `s3` *will result in an implicit redistribution of* `A` *to a replicated representation since the corresponding dummy argument is not distributed. A similar implicit redistribution after return from* `SUB2` *is performed to restore the original distribution of* `A`*.*

4 Independent Do Loops

The `INDEPENDENT` directive may be used in order to indicate to the system that a `DO` loop does not contain loop carried dependencies, and thus the loop iterations may be executed in any order.

Note:
VFC currently parallelizes `DO` loops only if they are preceded by an `INDEPENDENT` directive.

The following clauses may be specified within the `INDEPENDENT` directive:

- the `NEW` clause, to declare *private* variables.
- the `REDUCTION` clause, for reduction operations.
- the `ON HOME` clause, to specify the distribution of loop iterations.
- the `REUSE` clause with or without a logical condition, for *schedule reuse.*
- the `RESIDENT` clause with or without a list of array names, to avoid communication.

The `NEW` clause asserts that the named variables are private to each iteration of the loop and thus do not cause dependences. The `REDUCTION` clause asserts that the named variables are updated in the `INDEPENDENT` loop only in reduction operations, which are subject to syntactic constraints, and which do not require a particular order to determine the result of the reduction.

For a detailed description of the semantics of private variables and reduction operations please consult the HPF or HPF+ Language Specification. A independent loop with a new clause and a reduction clause is shown in Example 4.1. The `ON HOME`, `REUSE`, and `RESIDENT` clauses are described in the next Sections.

Example 4.1: Private Variables

```
!HPF$ INDEPENDENT, NEW(NN1,...), REDUCTION(F)
      DO I = 1, NUM
         NN1 = IX(1,I)
         ...
         DO J = 1, 6
            F(J,NN1) = F(J,NN1) + G(J,I)
            ...
         END DO
      END DO
```

This example shows an `INDEPENDENT` *directive with a* `NEW` *clause and a* `REDUCTION` *clause. Note that the variable* `F` *may only be used in reduction statements that are according to the syntax of reduction statements as described in the Appendix.*

4.1 On Clauses

By means of the ON HOME clause the user may specify how the iterations of an INDEPENDENT loop should be distributed. The ON HOME clause may be used to specify that a loop iteration should be executed by the processor that owns a particular array element. Note that the variable referenced in the ON HOME clause must be an array variable that references in exactly one subscript the loop iteration variable of the corresponding *do-stmt*. In case of nested independent loops it is required by VFC that the variable specified in the ON HOME clause of an inner loop is a subarray or a subelement of all variables specified in the ON HOME clause of surrounding independent loops. A nested independent loop with on-clauses is shown in Example 4.2.

Example 4.2: Nested Independent Loop with ON HOME clauses

```
!HPF$ INDEPENDENT, NEW(j), ON HOME(A(I,:))
      DO I = L1, U1
!HPF$    INDEPENDENT, ON HOME(A(I,J))
         DO J = L2, U2
            A(I,J) = A(X(i),j) + A(I,Z(J))
         END DO
      END DO
```

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

4.2 Resident Assertion

The RESIDENT clause may be used within an INDEPENDENT directive in combination with the ON HOME clause to specify that the data accessed in the corresponding loop is local to the executing processors, and thus no communication is required. If the resident-clause includes a list of arrays, then the locality assertion refers only to the specified arrays. This information is utilized to avoid the generation of unnecessary communication.

Example 4.3: The RESIDENT Clause

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

This example shows a loop from the FIRE HPF+ benchmark kernel. The RESIDENT *clause asserts to the compiler that all accesses to the distributed* PS *are local if the work distribution of the loop is determined according to the specified* ON HOME *clause, i.e. loop iteration* I *is executed on the owner of element* LB(I)*. Thus, no communication is required. Note that due to the indirect data accesses such an information cannot be derived automatically by the compiler.*

4.3 Schedule Reuse

VFC supports the reuse of communication schedules required for accesses to distributed array within `INDEPENDENT` loops by means of the `REUSE` clause. If the `REUSE` clause is specified within an `INDEPENDENT` directive, this constitutes an assertion to the compiler that the communication schedules of all arrays accessed in the loop are invariant for *all* executions of the loop and thus have to be computed only once. By specifying the `REUSE` clause the user asserts that

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

Note that this also applies to arrays referenced in the `ON HOME` clause, which must be present if the `REUSE` clause is specified.

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

Example 4.4: The `REUSE` Clause

```
!HPF$ DISTRIBUTE (BLOCK) :: A, B, C, D
      ...
      DO T = 1, MAX_TIME                                    ! time-step loop
         CALL SUB(A, B)

!HPF$    INDEPENDENT, ON HOME(A(...)), REUSE(MODULO(t,10)/=1)
         DO J = 1, N
            C(...) = D(...)
         END DO
      END DO
      ...

SUBROUTINE SUB(X, Y)
!HPF$ DISTRIBUTE (BLOCK) :: X,Y
      ...
!HPF$ INDEPENDENT, ON HOME(X(...)), REUSE
      DO I = 1, N
         X(...) = Y(...)
      END DO
END SUBROUTINE SUB
```

The `REUSE` *clause specified for the* `I`*-loop constitutes an assertion to the compiler that the communication schedules of* `A` *and* `Y` *are invariant for all invocations of subroutine* `SUB`*. Thus these schedules are computed once when* `SUB` *is called for the first time and reused in all other executions of* `SUB`*. The condition associated with the* `REUSE` *clause of the* `J`*-loop implies that the communication schedule for* `C` *and* `D` *will be (re)computed in iterations* $1, 11, 21, ...,$ *of the outer time-step loop.*

4.4 Purest Procedures

The `PUREST` directive may be specified for procedures that exclusively operate on local data. This information can be exploited by the compiler for an efficient parallelization of `INDEPENDENT` loops that contain procedure calls.

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

Note:
VFC currently requires that any procedure that is invoked from within an independent loop is *purest*.

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

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

Example 4.5 which is an excerpt of one of the HPF+ kernels shows the use of purest procedures in the context of independent loops.

Example 4.5: Purest Procedures

```
!HPF$ DISTRIBUTE (*,BLOCK) :: x,ix
      ...

!HPF$ INDEPENDENT, NEW(XN, F)
      DO I = 1, N
         XN = X(:,IX(:,I))     ! assign to private array
         CALL SUB(F, XN)    ! call purest procedure
         ...
      END DO
      ...


      PURE SUBROUTINE SUB(F, X)
!HPF$ PUREST
        ...
      END SUBROUTINE SUB
```

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

4.5 Restrictions on the Use of Independent Loops

All `INDEPENDENT` loops are currently parallelized by applying the *inspector/executor strategy.* This version of the system imposes a number of restrictions on the use of distributed arrays within such loops (see below). Whenever one of these restrictions is violated an `INDEPENDENT` loop might be sequentialized. If this is the case, it is indicated to the user by means of a *diagnostic message.*

The parallelization of `INDEPENDENT` loops is currently subject to the following restrictions:

- The bounds-expressions of loops that are nested within `INDEPENDENT` loops must not reference outer loop variables or distributed objects.
- `INDEPENDENT` loops and any loops that are nested within an `INDEPENDENT` loop must not have a step-size other than 1.
- Procedures called from within an `INDEPENDENT` loop must be `PUREST`.
- The use of sections of distributed arrays is not yet supported within `INDEPENDENT` loops.
- The `SELECT`, `REDISTRIBUTE`, `ALLOCATE` and `DEALLOCATE` statements are not yet supported within `INDEPENDENT` loops.

5 Extrinsic Procedures

The HPF_LOCAL extrinsic mechanism may be used to switch between global and local programming models. The use of HPF_LOCAL extrinsic procedures provides an easy means to integrate, for example, those parts of a sequential code that may be used in an SPMD style with distributed data but without the need for communication. Another application of HPF_LOCAL procedures are special forms of parallel I/O, for example, if each processor has to read from, or write to, a local file exactly those parts of a distributed array that are owned by it.

VFC currently only implements the argument transfer mechanisms associated with HPF_LOCAL procedures but does not yet support the HPF_LOCAL library. Calls to HPF_LOCAL procedures are transformed such that for a distributed argument on each processor only the local section that is owned by this processor is transferred. The code of an HPF_LOCAL extrinsic procedure is not modified by the compiler and thus no overhead is introduced.

Calls to HPF_SERIAL procedures are executed by one processor only. Prior to calling such a procedure, distributed actual arguments are redistributed by the system to a replicated representation. After return, the original distribution of actual arguments is restored. Moreover, a broadcast operation may be performed by the system for scalar arguments that may have been changed within an HPF_SERIAL procedure.

Note: Extrinsic procedures as supported by VFC must not contain any HPF directives.

Example 5.1: Extrinsic Procedures

```
!HPF$ PROCESSORS R(10)
!HPF$ DISTRIBUTE (BLOCK) :: A(100)

      INTERFACE
          EXTRINSIC (HPF_LOCAL) SUBROUTINE LOCSUB(X)
             REAL, DIMENSION(::) :: X
          END SUBROUTINE LOCSUB

          EXTRINSIC(HPF_SERIAL) SUBROUTINE SERSUB(X)
            REAL :: X(:), S
          END SUBROUTINE SERSUB
      END INTERFACE

      CALL LOCSUB(A)

      CALL SERSUB(A, S)
```

This example shows the use of extrinsic procedures. The call to the HPF_LOCAL *subroutine* LOCSUB *is executed by all processors. On each processor, however, only those parts of the actual argument* A *that are owned by the executing processor are transferred to the corresponding formal argument* X. *For example, on processor* R(5) *the array section* A(51:60) *is transferred to the formal argument* X. *The call to the* HPF_SERIAL *subroutine* SERSUB *is executed by one processor only. Prior to the call of* SERSUB *array* A *is implicitly redistributed to a replicated representation. After return from* SERSUB *the original distribution of* A *is restored and the scalar argument* S *is broadcast in order to ensure that* S *has the same value on each processor since it might have been modified within* SERSUB.

6 Compiling and Executing HPF+ Programs

HPF/HPF+ programs are compiled using the command `vfc`, which invokes a script that calls the VFC compiler, then the F90 compiler, and finally the linker, automatically linking in the necessary runtime libraries. The result of the compilation, if successful, is an executable file.

6.1 Command Line Syntax

The VFC command-line syntax is

vfc [*options*] *filename* ...

Where:

options	is zero or more command-line options (see below).
filename	is the name of a source file. More than one filename may be specified.

Options is zero or more of:

`-f`	free source form
`-i` *path*	Add *path* to the list of directories in which to search for `INCLUDE`-files or `MODULE`s.
`-m[cils]`	Insert instrumentation calls. `c` ... instrument communication statements `i` ... instrument `INDEPENDENT DO` loops `l` ... instrument sequential `DO` loops `s` ... instrument `SUBROUTINE` calls If `m` is used at least one of the options `[cils]` must be specified.
`-o` *out*	Name of output files or output file directory (depending on `-s` option). If `-o` is specified but not `-s` the final executable file is named *out* and the intermediate Fortran 90 message passing program *out*`_vfc.f90` instead of `a.out` and `a_vfc.f90`. If `-o` is specified together with `-s`, the intermediate message passing programs are generated in the directory *out* (see below).
`-s`	Stop after generation of the intermediate message passing Fortran 90 code. For each input file *name.ext* an output file with name *name*`_vfc.f90` is generated in the current directory, or in the directory *out* if `-o` *out* has been specified.
`-b` *n*	Specify buffer allocation ratio. *n* is a positive integer number. It should be set according to allocation ratio, if a program stops with the message: "DVIEW: not enough space in buffer - allocation ration *n*".
`-d`	Dynamic buffer management. (Should be used when a program stops with the message: "DVIEW: not enough space in buffer")
`-h`	Enable the use of halo areas.

6.2 Compiling HPF+ Programs

HPF+ programs are compiled by means of the `vfc` command as described in Section 6.1. During compilation the following types of messages may be issued:

- error messages
- restriction messages
- diagnostic messages

An *error message* is either a syntax error or a semantics error. A *restriction message* is issued if certain language features that are not yet fully supported by VFC are used. A *diagnostic message* informs the user about potential problems encountered during the parallelization process. In the presence of error or restriction messages no intermediate SPMD code or executable program is generated. Diagnostic messages, however, do not result in a failure of the usual compilation process, however, the performance of the generated code may be severely impaired.

VFC is currently available only on the QSW CS-2, whereas the VFC runtime system has been ported to different parallel architectures (and Fortran 90 compilers) including the CS-2, the NEC Cenju-3 and clusters of workstations.

If `vfc` is not found in your environment on the CS-2 you have to execute the following command:

```
source /opt/local/etc/vfcs.cshrc
```

6.3 Executing programs parallelized with VFC on the CS-2

The generated executable program may be run on the Meiko by means of the `prun` or `mpirun` commands. For example, if a program has been parallelized by VFC with the command

```
vfc -o myprog -f myprog.hpf
```

it can be executed on the Meiko (e.g. on 16 processors) using the following `prun` command:

```
prun -p p16b -n 16 myprog
```

Please consult the corresponding man-pages or related documentation for more information.

6.4 Executing programs parallelized with VFC on the Cenju-3

On the NEC Cenju-3 only the VFC runtime system is currently available, but not the VFC compiler. Thus the parallelization has to be done elsewhere, and the VFC generated SPMD F90 program with the name *out*`_vfc.f90` has to be transferred to the Cenju-3. There it has to be compiled and linked with the runtime system and the MPI libraries as follows:

```
make -f /tmp_mnt/home/sigi/VFC/vfc.Makefile T=out
```

6.5 Performance Measurement

If the system is called with the `-m` option followed by at least one of the characters 'c', 'i', 'l', or 's', the corresponding program constructs are instrumented by inserting calls to a performance measurement system. The meaning of the characters of the `-m` option is as follows:

- 'c' ... instrument communication routines
- 'i' ... instrument `INDEPENDENT DO` loops
- 'l' ... instrument `DO` loops that are not preceded by `INDEPENDENT`
- 's' ... instrument `SUBROUTINE` calls

In the case the `-m` option is specified, the following additional files are generated by VFC.

- `mdesc` (measurement description file)
- `mpi_file.c` (MPI register file)

After successful execution of the parallel program a trace file with the name *out*`_profile.log` or `a.out_profile.log`, if no output file name has been specified, is generated.

7 Restrictions

This section lists those features of HPF-2 which are not yet supported in this version of VFC. Most of these restrictions will be removed in future versions.

Note that VFC currently does neither provide special optimizations for the parallelization of regular `DO` loops nor for the automatic parallelization of loops without an `INDEPENDENT` directive. The main focus so far has been on the parallelization of codes which are characterized by independent loops exhibiting indirect, possibly dynamically changing, array accesses. *All* `INDEPENDENT` loops are currently parallelized based on runtime-compilation techniques. Consequently, a large overhead may be introduced for loops that do not contain indirect array references. The same is true for `INDEPENDENT` loops that are sequentialized because some restrictions are violated.

- Data distribution
  - The distribution of objects of type `CHARACTER`, `LOGICAL`, and `COMPLEX` is not yet supported.
  - The distribution of objects of derived types and components of derived types is not yet supported.
  - `TEMPLATES` are not supported.
  - Only *basic alignments* are supported; i.e. each *align-source* must be an *align-dummy* or '*', each align-subscript must be a *align-dummy* or '*', and the *align-target* must not have the `ALIGN` attribute.
  - *Prescriptive*, *descriptive* and *transcriptive* mappings for dummy arguments as specified in HPF are not supported.
  - The HPF intrinsic and library procedures are not yet supported, with the exception of `NUMBER_OF_PROCESSORS()`.
  - A distributed array must not have the `SAVE` attribute.

- The parallelization of `INDEPENDENT` loops is currently subject to the following restrictions:
  - The bounds-expressions of loops that are nested within `INDEPENDENT` loops must not reference outer loop variables or distributed objects.
  - `INDEPENDENT` loops and any loops that are nested within an `INDEPENDENT` loop must not have a step-size other than 1.
  - Procedures called from within an `INDEPENDENT` loop must be `PUREST`.
  - The use of sections of distributed arrays is not yet supported within `INDEPENDENT` loops.
  - The `SELECT`, `REDISTRIBUTE`, `ALLOCATE` and `DEALLOCATE` statements are not yet supported within `INDEPENDENT` loops.

  Note: Whenever one of these restrictions is violated an `INDEPENDENT` loop might be sequentialized, but the usual compilation process is not terminated. If this is the case, it is indicated to the user by means of a *diagnostic message*.

- Other HPF features
  - The HPF extrinsic library is not yet supported.
  - The FORALL statement and FORALL construct is not yet supported.
  - The HPF SEQUENCE directive is not supported.

- Fortran 77 features
  - Implied DO list I/O for distributed objects is not supported.
  - The EQUIVALENCE statement for distributed objects is not supported.
  - Statement functions are not supported.
  - Arithmetic IF statements are not supported.
  - Specifying different distributions in different program units for objects of COMMON blocks is not supported.

- Fortran 90 features
  - The WHERE statement and construct are not yet supported if they reference distributed objects.
  - The pointer assignment statement for distribute objects is not yet supported.
  - Array construction functions (PACK and UNPACK) are not yet supported for distributed arrays.
  - The ALL, ANY, COUNT, CSHIFT, DOT_PRODUCT, EOSHIFT, MATMUL, MAXLOC, MAXVAL, MINLOC, MINVAL, MERGE, RESHAPE, SPREAD, TRANSFER, and TRANSPOSE intrinsic functions are not yet supported for distributed arrays.
  - Distributed objects must not be referenced in *specification expressions*.
  - The transfer of sections of distributed arrays (other than whole arrays) to procedures is not yet supported.

References

[1] ANSI. *American National Standard Programming Language FORTRAN 77.* Number X3.9-1978. American National Standards Institute, 1978.

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

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

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

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

[6] S. Benkner, E. Laure, H. Zima. Final Definition of HPF+. Deliverable D2.1c of the ESPRIT IV LTR Project HPF+. University of Vienna, February 1998.

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

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

[9] High Performance Fortran Forum. *High Performance Fortran Language Specification. Version 1.0*, May 3, 1993.

[10] High Performance Fortran Forum. *High Performance Fortran Language Specification. Version 2.0*, High Performance Fortran Forum, January 31, 1997.

[11] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 1.1, June 1995. MPI-2: Extensions to the Message-Passing Interface, July 1997.

[12] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongara, MPI: The Complete Reference, MIT Press, 1996.

A Syntax

In the following the syntax of HPF+ and HPF language features as currently implemented in VFC is specified.

A.1 Syntax of Directives

hpf-directive-line	**is**	*directive-origin hpf-directive*
directive-origin	**is**	`!HPF$`
	or	`CHPF$`
hpf-directive	**is**	*specification-directive*
	or	*executable-directive*
specification-directive	**is**	*processors-directive*
	or	*distribute-directive*
	or	*align-directive*
	or	*dynamic-directive*
	or	*inherit-directive*
	or	*range-directive*
	or	*halo-directive*
	or	*purest-directive*
	or	*combined-directive*
executable-directive	**is**	*independent-directive*
	or	*redistribute-directive*
	or	*update-halo-directive*

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

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

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

Constraint: An *hpf-directive-line* follows the rules of Fortran fixed form (3.3.2.1) comment lines.

A.2 Combined Directives

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

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

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

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

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

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

A.3 PROCESSORS Directive

processors-directive	**is**	`PROCESSORS` *processors-decl-list*
processors-decl	**is**	*processors-name* [`(` *explicit-shape-spec-list* `)`] [`=` *associate-procs*]
associate-procs	**is**	*processors-name*
	or	`RESHAPE (` *processors-name* `)`

A.4 DISTRIBUTE and REDISTRIBUTE Directives

distribute-directive	**is**	`DISTRIBUTE` *distributee dist-attribute-stuff*
redistribute-directive	**is**	`REDISTRIBUTE` *distributee dist-attribute-stuff* [, *halo*]
	or	`REDISTRIBUTE` *dist-attribute-stuff* [, *halo*] :: *distributee-list*
dist-attribute-stuff	**is**	*dist-format-clause* [*dist-onto-clause*]
distributee	**is**	*object-name*
dist-format-clause	**is**	(*dist-format-list*)
dist-format	**is**	`BLOCK` [(*int-expr*)]
	or	`CYCLIC` [(*int-expr*)]
	is	`GEN_BLOCK` (*int-array-expr*)
	or	`INDIRECT` (*int-array-expr*)
	or	*
halo	**is**	`HALO` *halo-attr-stuff*
dist-onto-clause	**is**	`ONTO` *processors-name*

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

Constraint: A *distributee* must not appear as an *alignee* in an `ALIGN` directive.

Constraint: A *distributee* must not have the `SAVE` attribute.

Constraint: An *object-name* mentioned as a *distributee* in a `REDISTRIBUTE` directive must have the `DYNAMIC` attribute.

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

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

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

Constraint: Any *int-array-expr* appearing in a *dist-format* must be one-dimensional.

A.5 ALIGN Directive

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

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

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

Constraint: An *align-target* must have the `DISTRIBUTE` or `DYNAMIC` attribute.

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

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

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

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

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

A.6 HALO Directive

halo-directive	**is**	`HALO` *distributee halo-attr-stuff*
halo-attr-stuff	**is**	(*halo spec-list*) `ON` *halo-proc-ref*
halo-spec	**is**	*int-array-expr*
	or	*
halo-proc-ref	**is**	*processors-name* (*halo-dummy-list*)

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

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

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

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

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

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

A.7 INHERIT Directive

inherit-directive	**is**	`INHERIT` *inheritee-list*

Constraint: An *inheritee* must be a dummy argument.

Constraint: An `inheritee` must not have the `DISTRIBUTED` attribute, the `ALIGN` attribute, the `DYNAMIC` attribute, or the `SAVE` attribute.

A.8 DYNAMIC Directive

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

Constraint: An object in `COMMON` may not be declared `DYNAMIC`. (To get this kind of effect, Fortran 90 modules must be used instead of `COMMON` blocks.)

Constraint: An object that has the `DYNAMIC` attribute must not have the `SAVE` attribute.

A.9 RANGE Directive

range-directive	**is**	`RANGE` *object-name range-attr-stuff*
range-attr-stuff	**is**	*range-distribution-list*
range-distribution	**is**	(*range-dist-format-list*) [*dist-onto-clause*]
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 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* or *int-array-expr* appearing in a *range-dist-format* within a *specification-directive* must be a specification expression.

A.10 The INDEPENDENT Directive

independent-directive	**is**	`INDEPENDENT` [*independent-clause-list*]
independent-clause	**is**	*new-clause*
	or	*reduction-clause*
	or	*on-clause*
	or	*reuse-clause*
	or	*resident-clause*
new-clause	**is**	`NEW` (*variable-list*)
reduction-clause	**is**	`REDUCTION` (*reduction-variable-list*)
reduction-variable	**is**	*array-variable-name*
	or	*scalar-variable-name*
on-clause	**is**	`ON HOME` (*variable*)
reuse-clause	**is**	`REUSE` [(*scalar-logical-expr*)]
resident-clause	**is**	`RESIDENT` (*resident-variable-list*)

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

Constraint: A *variable* named in the `NEW` option must not be a dummy argument.

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

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

A.11 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**	+
	or	-
	or	*
	or	/
reduction-function	**is**	MAX
	or	MIN

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*.

A.12 PUREST Directive

purest-directive **is** `PUREST`

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

A.13 UPDATE_HALO Directive

update-halo-directive **is** `UPDATE_HALO ::` *object-name-list*

Constraint: Any *object-name* mentioned in the *update-halo-directive* must be a the name of a distributee or the name of an alignee.

A.14 Function and Subroutine Statements

function-stmt	**is**	[*prefix*] `FUNCTION` *function-name* ([*dummy-arg-name-list*]) [`RESULT` (*result-name*)]
subroutine-stmt	**is**	[*prefix*] `SUBROUTINE` *subroutine-name* [([*dummy-arg-list*])]
prefix	**is**	*prefix-spec* [*prefix-spec*] ...
prefix-spec	**is**	*type-spec*
	or	`RECURSIVE`
	or	`PURE`
	or	`HPF_LOCAL`
	or	`HPF_SERIAL`

Constraint: Within any HPF *external-subprogram*, every *internal-subprogram* must be of the same extrinsic kind as its host and any *internal-subprogram* whose extrinsic kind is not given explicitly is assumed to be of that extrinsic kind.