VFC - A Compiler for HPF+

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

Deliverable: D2.2c
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

In this report we describe the Vienna HPF+ Compiler (VFC), a new parallelizing compiler for HPF+, which has been developed in the context of the HPF+ project.

VFC is a command-line parallelization system that translates Fortran90/HPF+/HPF programs to Fortran90/MPI message passing programs. The system is currently available for the Solaris operating system. Codes parallelized with VFC can be executed on the QSW CS-2, the NEC Cenju3 and on clusters of workstations. VFC can be ported to any parallel system that provides a full Fortran 90 compiler and an MPI implementation.

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. To our knowledge, VFC is the first HPF compilation system that offers besides the basic block and cyclic data distributions, new data distribution formats required for irregular codes such as the general block distributions and indirect distributions, and that fully implements dynamic data redistribution.

Whereas the predecessor of VFC, the VFCS system [11], which has been used in the initial phase of the HPF+ project was based on Fortran 77 with support for some Fortran 90 extensions, VFC accepts full Fortran 90 and removes most of the restrictions of VFCS. Moreover, VFC is a fully automatic compiler which, opposed to its predecessor VFCS, does not require any user interaction to steer the parallelization process.

In order to provide support for dynamic memory allocation, dynamic distribution/redistribution, parameterization of programs by the number of processors, communication schedule reuse, separate compilation, and others, a completely new parallelization methodology including a new runtime support system has been developed and implemented.

The provision of these important new features, which were not supported by VFCS, and the fact that VFC is a completely new development, rather than an extension or enhancement of VFCS, motivated the decision to choose a new name for the compiler.

In the following we describe the overall structure of VFC, the supported language features, the main parallelization techniques including the runtime support system, and describe the use of the system and the process of executing programs parallelized by VFC on parallel computers.

The rest of this report is structured as follows. Section 2 gives an overview of VFC. Section 3 describes the HPF and HPF+ language features that have been implemented. Section 4 describes the compilation techniques of VFC with a special focus on techniques for the parallelization of irregular applications. Section 5 gives an overview of the runtime support system. Section 7 describes how VFC may be used to parallelize and execute HPF+ programs. Section 8 summarizes the major achievements and contributions of the work performed in the context of developing VFC. The syntax of the implemented HPF+ language features can be found in the Appendix.

2 Overview of VFC

In this chapter we give an overview of VFC and outline the structure of the system and the major compilation techniques.

2.1 Major Aspects of VFC

VFC performs a source-to-source translation from HPF/HPF+ programs to explicitly parallel Message Passing Fortran-90 programs. The parallelization techniques of VFC are based upon the *Single-Program-Multiple-Data (SPMD)* paradigm. The generated SPMD program utilizes calls to the MPI library to implement communication between the processors of a parallel machine and thus can be run on any platform that provides an MPI implementation and a Fortran 90 compiler. Currently the system is available on the QSW CS-2 and on clusters of SUN workstations. The VFC runtime system is available on the QSW CS-2, the NEC Cenju-3, and on a cluster of Solaris workstations.

Input languages for VFC are HPF+ and HPF-2 [29] with most of the language features specified in the HPF-2 Approved Extensions.

VFC provides very general and flexible parallelization strategies which are applicable also to programs with dynamically allocated and/or dynamically distributed arrays. Programs may be parallelized for an arbitrary number of processor which does not need to be fixed at compile time. The support for important modularization features like procedure interfaces, internal procedures, and modules significantly alleviates the parallelization of large programs.

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. Loops with irregular runtime dependent access patterns are essential for advanced industrial applications and in particular for the benchmark programs developed with the HPF+ project, but are currently not adequately supported by commercial HPF compilers. 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 different variants of the loop parallelization strategy are applied. A completely different 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 Section 4.

VFC implements, apart from some minor restrictions, all features for the transfer of distributed arguments across procedure boundaries and supports the `HPF_LOCAL` and `HPF_SERIAL` interfaces.

In order to support analysis and tuning of parallelized programs VFC provides integrated support for program instrumentation. The program instrumentation system allows the user via command-line options to automatically instrument the parallel program for obtaining detailed information about the execution of the parallelized code. The main focus of the instrumentation is on generating MPI trace-files for the MEDEA system providing detailed execution time profiles for the different phases of the loops that have been parallelized with the inspector/executor strategy.

2.2 Data Parallel Language Features

VFC supports full Fortran 90/HPF-2, most of the HPF+ language features and a number of features from the HPF-2 Approved Extensions. A detailed description of the implemented HPF and HPF+ language features can be found in Section 3 and in the Appendix.

Processor Arrays

Arbitrary multidimensional processor arrays may be defined. The program may be parallelized for an arbitrary number of processors by using the `NUMBER_OF_PROCESSORS()` intrinsic function in processor declarations. A relationship between different processor arrays may be established.

Data Distribution Features

VFC supports all the basic data distribution mechanisms (`BLOCK, CYCLIC`, block-cyclic distributions, replication, alignment), and in addition general block and indirect distributions as proposed in HPF+ and adopted by the HPF-2 Approved Extensions. Dummy arguments may either be explicitly distributed using the above mechanisms or may inherit their distribution from the corresponding actual arguments.

The set of non-local elements of an array that may be accessed on a particular processor can be specified by means of the `HALO` attribute at the time the distribution of an array is specified. This information is utilized by the compiler to determine the required communication schedules for updating the halo areas of an array. The `UPDATE_HALO` directive enables the user to specify when the halo area of an array should be updated.

All distributions mentioned above may be defined also for *dynamically allocated* objects. Moreover, VFC supports *dynamic data redistribution* either by means of the `REDISTRIBUTE` directive or redistribution as the result of procedure calls.

For a dynamically distributed array or for an array with the `INHERIT` attribute the range of different distributions the array may be associated with may be specified by means of the `RANGE` attribute.

VFC provides the following data distribution features:

- *Regular block and cyclic distributions*
- *General block and indirect distributions*
- *Alignments*
- *Halos*
- *Inheritance of distributions*
- *Distribution of* `ALLOCATABLE` *and* `POINTER` *objects*
- *Dynamic distributions and redistribution*
- *Distribution ranges*

Independent Loops

The INDEPENDENT directive may be used to assert to the compiler that the iterations of the subsequent DO loop are independent and may be executed in any order. In the context of the INDEPENDENT directive VFC supports the NEW clause for defining variables that are private to a loop iteration, the REDUCTION clause for performing reduction operations, the ON HOME clauses which enables the user to specify how the loop iterations should be distributed, the REUSE clause for indicating that the communication schedules of a loop can be reused (possibly depending on a logical condition), and the RESIDENT clause to avoid the generation of unnecessary communication. A detailed description of INDEPENDENT loops as currently implemented in VFC is given in Section 3.8.

Other Features

VFC provides *purest procedures* to be called from INDEPENDENT loops and which operate exclusively on data that is private to a processor. By specifying the PUREST attribute the user asserts to the compiler that the called procedure is PURE (free of side effects) and its invocation does not result in communication on entry, during activation, and upon exit from the procedure. This information, which is usually very hard to determine automatically, significantly supports the compiler in generating efficient code for independent loops that contain procedure calls.

VFC supports the extrinsic interfaces HPF_LOCAL and HPF_SERIAL. Whereas the associated argument transfer mechanisms for these extrinsic have been implemented, the HPF_LOCAL library is not yet available.

2.3 Fortran 90 Language Features

In the following we summarize important Fortran 90 features offered by VFC. Note that none of these features have been supported by the predecessor system VFCS.

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 directly, by means of 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.

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.

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 arguments and the transfer of sections of distributed 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.

The distribution of dummy arrays may be specified either with the `INHERIT` attribute or explicitly by means of the `DISTRIBUTE` or `ALIGN` attribute. In the latter case, if the distribution is specified explicitly, the actual and dummy argument must have the same distribution.

Input/Output

For the I/O of non-distributed objects VFC supports all the input/output mechanisms of Fortran 90, with some restrictions for data of type `COMPLEX`, `LOGICAL` and `CHARACTER`. 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.

2.4 Overview of the Parallelization Strategy

SPMD 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]. At runtime each processor of the target machine executes the same target program, but operates only on its local portion of the data.

Analysis of Distribution Directives

VFC analyzes the distribution specifications and generates for each distributed array calls to runtime routines that generate and initialize the corresponding runtime layout descriptors. Layout descriptors contain all information concerning the distribution of an array (or a class of arrays) together with global and local shape information and auxiliary information needed, for example, to allocate buffers for storing non-local data. Layout descriptors and the associated runtime routines have been designed in such a way that it is possible to use a single descriptor for arrays that have the same distribution. Moreover, it is possible to incrementally update layout descriptors, which significantly simplifies the implementation of dynamic memory allocation and dynamic data redistribution.

[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.

In order to implement the complex transfer mechanisms of distributed arguments at procedure boundaries, VFC analyzes all procedure calls and determines whether the distribution of an actual argument and the corresponding dummy argument are different, or if a distributed actual argument is associated with a non-distributed actual argument or vice versa. This analysis utilizes information from interface blocks (if present) with the goal to avoid implicit redistributions or copy-in/copy-out argument transfers whenever possible.

In cases where actual and formal distribution match or if the formal argument inherits the distribution, procedure calls are transformed such that each distributed argument is passed together with its layout descriptor without the need of redistribution or copying.

If, for example, a distributed actual argument is associated with a non-distributed formal argument, an implicit redistribution to replicate the actual argument is generated prior to the call, and after the call to restore the original distribution. In cases where such a redistribution is generated, or if arguments have to be copied, the user is informed of this potential runtime overhead by corresponding *diagnostic messages.*

Classification of Program Constructs

After data distribution analysis VFC analyzes the code in order to classify each program construct (e.g. array assignments, loops) that references distributed objects according to the parallelization strategy that will be applied to it.

Parallelization of Array Assignments

Depending on the distributions and access patterns of distributed arrays, VFC distinguishes three different classes of array assignments which are handled with varying overheads.

The first class of array assignments can be handled in an optimal way without the need for any transformations. This requires that each dimension of all arrays on the right-hand-side with a non-scalar subscript is distributed in exactly the same way as the corresponding dimension of the left-hand-side array and that all corresponding distributed dimensions of all arrays involved are accessed identically and as a whole.

The second class comprises array assignment statements for which only work distribution and index conversion has to be performed, but which can be processed without introducing inter-processor communication. This strategy is applied if, for all arrays on the right-hand-side, each dimension with a non-scalar subscript is distributed in exactly the same way as the corresponding dimension of the left-hand-side array. Such statements are transformed by adapting the section-subscripts of distributed dimensions in such a way that each processor only accesses its local data. These local section-subscripts are determined at runtime by intersection of the local index set of an array dimension with the original global section subscripts and converting the resulting section into local indices.

All other array assignment statements are transformed by introducing appropriate calls to runtime library routines that determine the work distribution and generate the required communication of non-local data into temporary arrays.

Unlike most HPF compilers, array statements are not converted into loops, but are transformed to array statements that operate on the local memory on each processor. The main advantage of this strategy is that the parallelism of the original construct is preserved. Therefore, both inter- and intra-processor parallelism can be exploited by the Fortran 90 compiler of the parallel target machine.

Parallelization of Independent Loops

The `INDEPENDENT` directive for loops enables the user to assert that the iterations are independent and can be executed in parallel. Furthermore, the work distribution (i.e. the distribution of loop iterations to processors) may be specified by the user. Parallelization of `INDEPENDENT` loops is implemented based on runtime compilation techniques. The potentially large overhead caused by preprocessing of such loops at runtime according to the inspector/executor strategy can be amortized in many cases by reusing communication schedules across subsequent executions of the loops. Reuse of communication schedules is supported by VFC based on the `REUSE` clause. Both unconditional and conditional schedule reuse have been implemented. The `RESIDENT` clause may be used to avoid unnecessary communication. Moreover, the `NEW` clause and the `REDUCTION` clause to support reduction operations including, summation, subtraction, multiplication, `MIN` and `MAX`, have been implemented.

Independent loops are currently parallelized based on the inspector/executor strategy. The compiler transforms such loops into three phases: the *work distributor*, the *inspector*, and the *executor*. On each processor, the work distributor calculates the set of loop iterations to be executed on that processor. The inspector performs a runtime analysis of the loop and generates a description of the communication schedule which is used by the runtime procedures that move data between processors in subsequent communication phases. Within the executor phase the actual computation is performed without the need for non-local data.

2.5 System Components

VFC has been developed based on a Fortran 90 frontend developed at GMD Berlin. The implementation languages used are C and C++. The Cocktail compiler construction toolbox has been extensively used. The internal representation of VFC, and in particular the structure of the abstract syntax tree, is formally specified based on the AST tool. A high-level approach has been adopted for implementing the various parallelization techniques based on the pattern-matching and tree-transformation tool PUMA.

The VFC system consists of six major components: the frontend, the analysis module, the parallelization module, the instrumentation system, the code generator, and the runtime system.

The frontend consists of the scanner, the parser and the tree-builder. These components have been implemented based on the Cocktail tools REX, LALR and AST, respectively. The frontend has been initially developed by GMD Berlin and has been extended within this project to support the HPF+ and HPF-2 language features. The internal representation of the program as constructed by the frontend consists of the abstract syntax tree, the (scoped) symbol table, the label table and a few other data structures. The intermediate representation is formally specified based on the AST tool. The semantic checker is to a large extent written in PUMA and performs various semantic checks, constant folding, resolving of overloaded procedures, and various other tasks.

The analysis module constructs different abstractions of the program such as the program dependence graph, the call graph and a flow graph for each program unit. These data structures are the basis for data dependence analysis and for an advanced interprocedural analysis framework [30].

The *parallelization module*, which transforms the input program into a parallel message passing SPMD program, consists of several components. The *normalization component* performs various transformations such as array and loop normalization, normalization of data distribution directives, substitution of function calls in expressions that reference distributed objects, normalization of procedure interfaces, and others. The task of the *classification component* is to determine the parallelization strategy that should be applied to each executable construct. A generalized inspector/executor strategy is utilized for parallelizing irregular loop nests. A parallelization technique based on intersection of regular sections is applied to regular loops and explicitly parallel constructs such as array assignments. A default strategy utilizing single-element send/receive/broadcast operations, is applied either to scalar assignments or to non-parallelizable loops. Depending on the availability of analysis information or the presence of user assertions, different variants of these strategies are applied. All parallelization strategies have been implemented based on the PUMA tool.

The *instrumentation component* inserts calls to a measurement library in order to generate MPI trace-files during execution of the program. The instrumentation can be triggered by the user by means of command-line options, and it supports selective program instrumentation, in order to provide the user with detailed information about the execution of the parallelized code.

The *code generator* translates the internal representation of the program into Fortran 90 source code, which is then submitted to the Fortran 90 compiler of the parallel target machine in order to generate the executable parallel program.

The runtime system of VFC consists of two parts: the VHPF and the PARS libraries. The main task of the VHPF library is the management of distributed objects at runtime, including runtime support for dynamic data redistribution. Moreover, the VHPF library provides high level interfaces for communication primitives of the PARS library calls which significantly alleviate code generation. The PARS library provides communication routines utilizing calls to the MPI-1 library [33] and supports all aspects of the inspector/executor runtime parallelization strategy, including the support for schedule reuse and halos.

The VFC compiler consists of about 225.000 lines of C/C++ code. More than 80% thereof are generated by the Cocktail tools from high-level specifications. In total, these high level specifications consist of approximately 85.000 lines of code, of which 60% are written in PUMA, 30% in AST, and the rest in REX and LALR. The runtime system is written in Fortran 90 and C and comprises about 20.000 lines of code.

3 HPF and HPF+ Language Features

In this section we describe the HPF and HPF+ language features that are currently supported by VFC. For a detailed specification of HPF+ please refer to the HPF+ language definition [8]. The syntax of the HPF+ language features supported by VFC is specified in the Appendix.

3.1 Processor Arrays

VFC allows the user to parallelize the program for an arbitrary number of processors using the `NUMBER_OF_PROCESSORS` intrinsic function in processor array declarations.

In the current version of VFC the basic HPF+ mechanisms for establishing relationships between processor arrays have been implemented. The intrinsic function `RESHAPE` may be used to specify that a processor array `Q` is a reshape of another previously declared processor array `R`. Correspondence between `Q` and `R` is established according to the Fortran array element order. An example of HPF+ processors directives is shown below.

Example 3.1: HPF+ Processors Directives

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

!HPF$ PROCESSORS Q(NUMBER_OF_PROCESSORS()/2,2) = RESHAPE(R)
```

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. Note that the number of processors the program can be executed on must in this case be at least 2 and should be an even number.*

VFC also supports the declaration of global processor arrays within modules and the declaration of local processor arrays within procedures.

3.2 Data Distribution Mechanisms

VFC supports arbitrary multi-dimensional distributions that may be specified either by using any combination of the block, cyclic, general block, and indirect distribution formats and replication, or by means of alignment. In addition to the distribution, the set of non-local elements accessed on a particular processor may be specified by means of the *halo attribute*. VFC fully implements the distribution of dynamically allocated objects including automatic arrays, and dynamic data distribution and redistribution.

3.2.1 Regular block distributions

A regular block distribution, `BLOCK`, partitions an array dimension to a processor dimension in evenly sized portions. Specifying a block-size explicitly (e.g. `BLOCK(M)`) is also supported.

3.2.2 Cyclic distributions

A cyclic distribution, `CYCLIC`, maps the array elements in round-robin manner to the processors. If `CYCLIC(M)` is used, contiguous blocks of size `M` are mapped in a round-robin manner to the processors.

3.2.3 General block distributions

A general block distribution, GEN_BLOCK, partitions an array into arbitrarily sized portions. These distributions are useful to deal with situations in which regular block distributions may lead to an unbalanced load, for example, if the boundary region of a data domain needs more work than the interior. General block distributions are also included in the HPF-2 Approved Extensions.

3.2.4 Indirect distributions

Indirect distributions enable the user to specify any arbitrary distribution that does not involve replication by means of a *mapping array*, which specifies for each array element the processor it should be mapped to. These distributions are also included in the HPF-2 Approved Extensions.

3.2.5 Replication

To indicate that an array dimension should be replicated, i.e. allocated on each processor, the "*" may be used.

3.2.6 Multidimensional Distributions

Arbitrary dimensions of multidimensional arrays may be distributed using any combination of block, cyclic, general block and indirect distributions and replication in the full generality as defined by HPF+ and HPF-2 with its Approved Extensions. As in HPF, the number of distributed dimensions must always be equal to the rank of the corresponding processor array (section).

Example 3.2: Static Distributions

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

!     multidimensional distributions
!
!HPF$ DISTRIBUTE(BLOCK,*,CYCLIC) ONTO R2 :: A3

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

!     indirect distributions
!
      INTEGER MAP(100)                  ! must be an array constant or dummy argument
!HPF$ DISTRIBUTE(INDIRECT(MAP),*) :: A2
```

This example shows statically distributed arrays. Note that all arguments of distribution formats of statically distributed array must be specification expressions.

3.2.7 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 in the sense that it cannot be destroyed at runtime. Thus alignment may be used to specify groups of arrays that are always distributed in the same way, even in the context of dynamic data distribution.

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

The other forms with an *align-source-list* and an *align-subscript-list* are provided to specify alignment with respect to individual array dimensions. Alignments as currently provided by VFC are shown in Example 3.3.

Example 3.3: HPF+ Alignment Directives

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

! 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,*)
```

3.3 Runtime-defined Static Distributions

VFC supports runtime defined distributions. This mechanism allows using arguments, which are defined at runtime, in the distribution formats of statically distributed arrays[2]. For example, this feature should be used if block-sizes of `BLOCK, CYCLIC` or `GEN_BLOCK` distributions 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

```
        INTEGER, DIMENSION(1000) :: MAP
        ...
        READ(*,*) MAP

        CALL SUB(MAP)
        ...

        SUBROUTINE SUB(MAP)
           INTEGER, DIMENSION(:) :: MAP

           REAL, DIMENSION(6,1000) :: A

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

           ...
        END SUBROUTINE SUB
```

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.*

3.4 Dynamic Memory Allocation

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.

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

[2]These are distributed arrays that do not have the `DYNAMIC` attribute.

Example 3.5: Dynamic Allocation of Distributed Arrays

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

!HPF$ PROCESSORS R(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE(BLOCK) :: A, P
...
ALLOCATE(A(N), STAT=...)

CALL SUB(P)
...

SUBROUTINE SUB(Q)
!HPF$ INHERIT :: Q
REAL, DIMENSION(:), POINTER :: Q
   ...
   IF (.NOT.ASSOCIATED(Q)) THEN
      ALLOCATE(Q(somesize), STAT=...)
      ...
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. Note that an explicit interface is required by Fortran for* `SUB` *since it has a dummy argument with the* `POINTER` *attribute.*

3.5 Dynamic Data 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. VFC supports dynamic redistribution in the full generality as specified by HPF+ and the HPF-2 Approved Extensions.

An *initial distribution* may be specified for arrays with the `DYNAMIC` attribute. If no initial distribution has been specified the array may not be accessed (or allocated) unless it has been distributed. Note that currently VFC does not allow to specify an initial distribution for a dummy argument that has the `DYNAMIC` attribute.

Data redistribution is supported without any restriction for multidimensional 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. Dynamic redistribution is also supported for arrays that are dynamically allocated. Note that in such cases an array must have been associated with a distribution prior to allocation as shown in Example 3.6.

Example 3.6: Dynamically Distributed Arrays

```
REAL, DIMENSION(N)             :: A
REAL, DIMENSION(N)             :: 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.*

3.6 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 an irregular, non-contiguous area describing the non-local access pattern for each processor. Halo regions are specified by means of the `HALO` attribute which may be combined with distribution specifications. 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.

The halo of a dimension of a distributed array is usually specified by means of 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. If an array `B` is *perfectly aligned*[3] with another array `A` that has a halo, `B` implicitly takes over the halo of the align-target `A`.

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 3.7.

Example 3.7: Using Halos

```
!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`*.*

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

3.7 The INDEPENDENT Directive

All HPF+ kernels include loops where distributed arrays are accessed by means of indirection arrays which are defined at runtime. All these loops do not contain loop carried dependences and thus can be executed in parallel. However, a compiler usually cannot determine the absence of dependences and thus will have to make worst-case assumptions which will lead to a sequential execution of such loops.

For this purpose, the `INDEPENDENT` directive is provided in order to enable the user to specify that a `DO` loop does not contain loop carried dependencies. The `NEW` clause may be used to define *private* variables, i.e. variables that are local to each loop iteration. The `REDUCTION` clause and *reduction statements* are provided in order to specify reduction operations. Currently the reduction statements with the operators **+,-,*,** and **/** and the reduction functions `MIN` and `MAX` are implemented.

All these language features are supported by VFC with the same syntax and semantics as defined in HPF-2. Example 3.8 shows an `INDEPENDENT` directive with a `NEW` clause and a `REDUCTION` clause.

Example 3.8: The INDEPENDENT Directive

```
!HPF$ INDEPENDENT, NEW(nn1,...), REDUCTION(f)
      do iel = 1, numel2
        nn1 = ix(1,iel)
        ...
        do idim=1,6
          f(idim,nn1) = f(idim,nn1) + force1(idim,iel)
          ...
        end do
      end do
```

This example shows an `INDEPENDENT` *directive with a* `NEW` *clause and a* `REDUCTION` *clause. The loop has been taken from the generic HPF+ PAM CRASH kernel which was developed by NEC.*

As an extension of the HPF `INDEPENDENT` directive, HPF+ provides a mechanism in order to specify the distribution of loop iterations. For this purpose 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. Moreover, the `RESIDENT` clause, optionally followed by a list of arrays, may be used together with the `ON HOME` clause to specify that no communication is required for the specified arrays, or for none of the arrays accessed in the loop, if `RESIDENT` is specified without a list of arrays.

Example 3.9: The `ON HOME` Clause and the `RESIDENT` Clause

```
!HPF$ PROCESSORS R(4)
      ...
!HPF$ INDEPENDENT, ON HOME(A(I)), RESIDENT(A)
      DO I=1,N
         A(I) = A(X(I)) + A(Y(I))
      END DO
```

In this example, the `ON HOME` *clause specifies that iteration* `I` *of the* `DO` *loop has to be executed on the processor that owns array element* `A(I)`*. The* `RESIDENT` *clause asserts that all accesses to* `A` *are already local and thus no communication is required for* `A`*.*

INDEPENDENT loops are transformed by VFC in such a way that in a preprocessing phase (inspector) for each accessed array a communication schedule is determined which describes for each processor the set of elements that are need from/by other processors for executing the loop. Building a schedule may be very expensive, in some cases dominating the execution time of the program. Often, this cost can be amortized across subsequent executions of a parallel loop, for example, if the parallel loop is enclosed in a sequential time-step loop. However, existing analysis techniques are not powerful enough to detect all instances of redundant schedule computations; runtime analysis techniques are similarly restricted.

Thus, VFC supports the reuse of communication schedules at the language level by means of the REUSE clause which may be specified in the context of INDEPENDENT loops. If the REUSE clause is specified, 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. In order to handle 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*.

Example 3.10: The REUSE Clause

```
      do t=1,max_time
!HPF$ INDEPENDENT, REUSE
         do i = 1, n
            ...
         enddo

!HPF$ INDEPENDENT, REUSE(MODULO(t,10)/=1)
         do j = 1, n
            A(...) = B(...)
         end do
      end do
```

The REUSE *clause specified in the* I*-loop constitutes an assertion to the compiler that the communication schedules for all arrays accessed within this loop are invariant. The condition associated with the* REUSE *clause of the* J*-loop implies that the communication schedule for* A *and* B *will be (re)computed in iterations* $1, 11, 21, ...,$ *of the outer loop.*

3.8 Purest Procedures

The PUREST directive enables the user to assert that a procedure may be called and executed without the need of communicating data between the processors. By specifying the PUREST directive for a procedure the user asserts that the invocation of the procedure does 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. To specify that a procedure is purest, the PUREST directive, which must occur in the specification part of a program unit, is provided.

3.9 Extrinsic Interfaces

VFC supports the extrinsic interfaces `HPF_LOCAL` and `HPF_SERIAL` and the associated argument transfer mechanisms. When an `EXTRINSIC (HPF_LOCAL)` procedure is called, on each processor only those parts of distributed arguments that are local are transferred to the procedure. On the other hand, when an `EXTRINSIC (HPF_SERIAL)` procedure is called distributed arguments are replicated before the call.

3.10 Fortran 90 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 [27].

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

4 Compilation

In this section we describe the compilation techniques implemented in VFC for the parallelization of HPF+ programs. VFC adopts a very general and flexible parallelization strategy supporting almost all features of HPF+, including arbitrary multidimensional distributions, alignments, dynamic memory allocation and dynamic data redistribution, schedule reuse, halos, and a number of other features.

The main focus within the HPF+ project has been on the parallelization of nested independent loops in the context of irregular, possibly dynamically changing, data distributions and access patterns. The parallelization techniques implemented in VFC are general in the sense that they are applicable to any HPF+ program with currently only a few restrictions. Note however, that due to the focus on parallelization techniques for irregular computations, no special optimizations for regular `DO` loops are currently available. Thus a large overhead may be introduced if the parallelization techniques for irregular computations are applied to programs that exhibit only regular distributions and access patterns. Efficient support also for highly regular constructs is available only for array assignment statements and is planned to be extended in future releases of the system.

In the following we describe the management and allocation strategy of distributed arrays, the transformation of processors declarations, the processing of distribution and alignment directives, the parallelization of array assignments and independent loops, the handling of dynamic data redistribution, the transformation of procedure calls and procedure interfaces and the implementation of the extrinsic interfaces.

4.1 Management of Distributed Arrays

Distributed arrays as supported by VFC may be statically or dynamically allocated and statically or dynamically distributed. A distributed array is either a *distributee* or an *alignee*. The distribution of arrays is usually parameterized with respect to the number of processors the program will be executed on based on the intrinsic function `NUMBER_OF_PROCESSORS`.

To support all these features, VFC transforms the input program in such a way that each distributed array is allocated at runtime at the earliest point at which both the shape information and the distribution information are available.

VFC implements a sophisticated runtime management strategy for distributed arrays based on runtime layout descriptors. Runtime layout descriptors contain all information concerning the distribution of an array (or a class of arrays) together with global and local shape information and auxiliary information needed, for example, to allocate buffers for storing non-local data, or for the generation or reuse of communication schedules.

After analyzing the distribution specifications, VFC generates for each distributed array calls to runtime routines that generate and initialize the corresponding *runtime layout descriptors*. Runtime layout descriptors have been designed such that it is possible to update them incrementally whenever the distribution or the shape of an array is modified at runtime.

The evaluation of the distribution of an array and the allocation of an array at runtime may have to be performed in separate steps depending on the properties of the array as outlined in the following sections.

4.1.1 Runtime Layout Descriptors

VFC generates for each distributed array a pointer to a *runtime layout descriptor* which stores all the information concerning the distribution of the array. Layout descriptors are manipulated at runtime whenever the shape or the distribution of an array is changed. The design and implementation of layout descriptors and the associated runtime system routines has been strongly influenced by the need to support the dynamic concepts of HPF+.

A runtime layout descriptor may consists of up to four main components,

- the distribution descriptor,
- the alignment descriptor,
- the memory descriptor,
- and the communication descriptor.

The *distribution descriptor* contains information concerning the distribution formats and the processor array to which an array is distributed. The *alignment descriptor* contains alignment information and a reference to the alignment-target array. The *memory descriptor* stores information about the global and local shape of the array as well as information needed for the storage management of non-local data. The *communication descriptor* contains information about communication schedules required for updating non-local data stored in buffers.

All layout descriptors of a particular program unit are managed by means of a list. Whenever a new descriptor for a distributed object has to be generated this list is searched in order to determine if a suitable descriptor already exists. If this is the case, no new descriptor is generated but only a pointer to a matching descriptor is returned. This is the basic mechanism for sharing of runtime layout descriptors in the context of dynamically distributed data objects. Furthermore, runtime tests to check whether two dynamically distributed arrays have the same distribution can be realized efficiently with this strategy.

4.1.2 Life-Cycle of Distributed Arrays

Statically distributed arrays have a fixed distribution within their scope of declaration. Thus, in the generated SPMD program, at the beginning of the executable section of the program unit where the array is declared, the distribution or alignment specifications are evaluated and the results are stored in the array's layout descriptor. The array is allocated dynamically on each processor according to its local shape stored in the memory component of the layout descriptor, either at the beginning of the executable section, if it was statically allocated in the HPF+ program, or, if it is dynamically allocated, at the time of allocation when an `ALLOCATE` statement is executed for it. This process is first carried out for all distributees and then for all alignees of the corresponding program unit. At the end of a program unit all distributed arrays which were statically allocated in the HPF+ program are deallocated. Note that for a statically distributed array the distribution component of the layout descriptor does not change, but the memory component may change if, for example, the shape of the array is modified.

Dynamically distributed arrays may change the distribution within their scope of declaration. Thus, whenever such an array is redistributed, the distribution component of the descriptor is updated and the array is reallocated with the new distribution, provided that its shape is known.

4.2 Transformation of PROCESSORS Directives

For each processors array declared in a program unit VFC generates at the beginning of the executable section a call to an initialization procedure which creates and initializes a *runtime processor array descriptor*. The initialization procedure, `vhpf_init_proc_dsc` takes a pointer to a processor array descriptor and the shape of the processor array as arguments, allocates the corresponding descriptor, and generates an MPI communicator for it.

Example 4.1: Transformation of PROCESSORS Directives

HPF processors statement ...

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

generated Fortran 90 code ...

```
      TYPE (proc_dsc), POINTER :: r_dsc   !  processor array descriptor
      ...
      CALL vhpf_init_proc_dsc(proc=r_dsc, shape=(/number_of_processors()/)
```

For each processors array declared in a program unit VFC generates at the beginning of the executable section a call to the runtime system routine `vhpf_init_proc_dsc` *to initialize the corresponding processor array descriptor.*

4.3 Transformation of Declaration Statements

The declaration statements of distributed arrays are transformed such that the arrays can be dynamically allocated in the SPMD target program. Depending on the distribution scheme and allocation scheme VFC distinguishes four different cases, which are described in the following.

4.3.1 Statically Distributed Arrays

For each statically distributed array the following transformations are performed:

1. The declaration of the array's *layout descriptor* is generated.
2. At the beginning of the executable section of the program unit where the array is declared, a call to the runtime system procedure `vhpf_init_rt_dsc` to initialize the layout descriptor with the distribution information and, if present, the shape information is inserted.
3. For a statically allocated array the following additional transformations are performed:
   (a) The `ALLOCATABLE` attribute is added to the declaration and the *explicit-shape-spec-list* is transformed into a *deferred-shape-spec* list.
   (b) After the call to `vhpf_init_rt_dsc` an `ALLOCATE` statement is generated, such that at runtime on each processor the local portion of the array is allocated based on the local shape information from the layout descriptor.
   (c) At the end of the corresponding executable section, a `DEALLOCATE` statement for the array is generated.

4.3.2 Dynamically Distributed Arrays

For each dynamically distributed array the following transformations are performed:

1. The declaration of the array's *layout descriptor* is generated.

2. At the beginning of the executable section of the program unit where the array is declared, a call to the runtime system procedure `vhpf_init_rt_dsc` to initialize the layout descriptor with the shape information and the initial distribution information, if present, is inserted.

3. For a statically allocated array the following additional transformations are performed:

   (a) The `POINTER` attribute is added to declaration statement and the *explicit-shape-spec-list* is transformed into a *deferred-shape-spec* list.

   (b) After the call to `vhpf_init_rt_dsc` an `ALLOCATE` statement is generated, which at runtime allocates on each processor the local portion of the array based on the local shape information from the layout descriptor.

   (c) At the end of the corresponding executable section, a `DEALLOCATE` statement for the array is generated.

4.4 Transformation of `ALLOCATE` Statements

The initialization of the layout descriptor for a dynamically allocated array[4] is done in two separate steps.

At the beginning of the program unit where the array is declared a call to `vhpf_init_rt_dsc` is generated resulting in a partial initialization of the layout descriptor with the distribution or alignment information if the array is statically distributed or dynamically distributed with an initial distribution.

Prior to an ALLOCATE statement of the array another call to the runtime system routine `vhpf_init_rt_dsc` is generated in order to update the shape component of the layout descriptor according to the global shape from the original allocate statement. At the time this routine is called, the distribution of the array is evaluated and on each processor the local shape of the array is computed. The array is then allocated with the local shape.

An `ALLOCATE` statement for a distributed array is transformed as follows:

1. A call to the library routine `vhpf_init_rt_dsc` with the shape information from the original `ALLOCATE` statement is inserted in front of the `ALLOCATE` statement.

2. The `ALLOCATE` statement is transformed such that on each processor only the local segment is allocated according to the local shape obtained from the layout descriptor by means of the runtime system function `vhpf_extent`.

This is shown in example 4.3.

[4]This is an array that has the `ALLOCATABLE` or `POINTER` attribute.

Example 4.2: Transformation of Statically Distributed Arrays

original declaration of statically distributed arrays ...

```
      REAL A(N), B(M,N)
!HPF$ PROCESSORS R(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE(INDIRECT(MAP)) ONTO R :: A   ! distributee
!HPF$ ALIGN (*,I) WITH A(I)           :: B   ! alignee

      ... ! first statement
```

generated Fortran 90 code ...

```
      REAL, ALLOCATABLE, DIMENSION(:)   :: a
      REAL, ALLOCATABLE, DIMENSION(:,:) :: b
      TYPE (rt_dsc), POINTER            :: a_dsc   ! runtime layout descriptor
      TYPE (rt_dsc), POINTER            :: b_dsc   ! runtime layout descriptor

      ! initialize layout descriptor for distributee A
      CALL vhpf_init_rt_dsc(dlist, a_dsc, 2, shape=(/n/), dtype=(/INDIRECT/), &
                            aarg1=(/map/), proc=r_dsc)

      ! allocate local segment of A
      ALLOCATE((a(1:vhpf_extent(a_dsc, 1)))


      ! initialize layout descriptor for alignee B
      CALL vhpf_init_rt_dsc(dlist, b_dsc, 2, shape=(/n/), alseq=(/0,1/), altrg=a_dsc)

      ! allocate local segment of B
      ALLOCATE((b(1:m, 1:vhpf_extent(b_dsc, 1)))

      ... ! first statement

      DEALLOCATE(b)
      CALL vhpf_free_rt_dsc(dlist, b_dsc)    ! free layout descriptor of B

      DEALLOCATE(a)
      CALL vhpf_free_rt_dsc(dlist, a_dsc)    ! free layout descriptor of A
```

This example shows the code generated for a statically allocated and statically distributed distributee `A` *and alignee* `B`*, respectively. A pointer to a runtime layout descriptor is generated for each array and the declarations of the arrays are changed such that they can be dynamically allocated. At the beginning of the executable section a call to the runtime system procedure* `vhpf_init_rt_dsc` *is generated which initializes the layout descriptor of* `A`*. After this call the local segment (including buffer space for non-local elements) of* `A` *is allocated on each processor according to the computed size which is stored in the descriptor and returned by the runtime system function* `vhpf_extent`*. Alignee B is handled similarly, except that instead of the distribution information the alignment information is passed to the initialization routine. At the end of the executable section of the corresponding program unit the local memory and the layout descriptors of both arrays are deallocated.*

Example 4.3: Transformation of Dynamically Allocated Distributed Arrays

original declaration of a dynamically allocated distributed array and `ALLOCATE` *statements*

```
      REAL, ALLOCATABLE :: A(:)
!HPF$ PROCESSORS R(NUMBER_OF_PROCESSORS())

!HPF$ DISTRIBUTE(CYCLIC) ONTO R :: A

      ...  ! first statement

      ALLOCATE(A(N), STAT=ALLOC_ERR)    ! s1
      ...

      DEALLOCATE(A)
      ALLOCATE(A(N*N), STAT=ALLOC_ERR)  ! s2
```

generated Fortran 90 code ...

```
      REAL, ALLOCATABLE, DIMENSION(:) :: a
      TYPE (rt_dsc), POINTER          :: a_dsc

      ! initialize descriptor with distribution info only
      CALL vhpf_init_rt_dsc(dlist, a_dsc, 2, dtype=(/CYCLIC/), proc=r_dsc)

      ...  ! first statement

      ! update descriptor with shape info from s1
      CALL vhpf_init_rt_dsc(dlist, a_dsc, 2, shape=(/N/),  proc=r_dsc)

      ! allocate local segment
      ALLOCATE((a(1:vhpf_extent(a_dsc, 1)), STAT=ALLOC_ERR)  ! s1
      ...

      DEALLOCATE(a)

      ! update descriptor with shape info from s2
      CALL vhpf_init_rt_dsc(dlist, a_dsc, 2, shape=(/N*N/),  proc=r_dsc)

      ! allocate local segment
      ALLOCATE((a(1:vhpf_extent(a_dsc, 1)), STAT=ALLOC_ERR)  ! s1
      ...
```

This example shows the transformation of `ALLOCATE` *statements. At the beginning of the executable section a call to* `vhpf_init_rt_dsc` *is generated in order to initialize the layout descriptor with the distribution information. Prior to each* `ALLOCATE` *statement another call to* `vhpf_init_rt_dsc` *is generated in order to update the shape information in the layout descriptor. The* `ALLOCATE` *statements are transformed such that instead of the global size only the local segment is allocated on each processor according to the local shape information from the descriptor which is accessed by means of the runtime system function* `vhpf_extent`*.*

4.5 Transformation of REDISTRIBUTE Statements

The layout descriptor of a dynamically distributed array[5] is usually initialized in two separate steps.

If the array does not have an initial distribution specified for it, then the layout descriptor initialization routine `vhpf_init_rt_dsc` is called at the beginning of the corresponding program unit with the shape information only (if present) resulting in a partial initialization of the layout descriptor.

The allocation of arrays without an initial distribution is delayed until the array has been associated with a distribution, which is usually done by executing a `REDISTRIBUTE` statement.

If a dynamically distributed array has an initial distribution and neither the `ALLOCATABLE` nor the `POINTER` attribute, its layout descriptor is fully initialized at the beginning of the executable section of the corresponding program unit and the array is allocated on each processor according to its local shape.

A `REDISTRIBUTE` statement is transformed as follows.

1. The declaration of a temporary layout descriptor for the array being redistributed is generated.
2. In front of the `REDISTRIBUTE` statement, a call to initialize the temporary descriptor with the new distribution information from the `REDISTRIBUTE` statement is generated.
3. A call to the runtime system routine `vhpf_redist`, which takes as arguments both the original and the temporary layout descriptor, is generated. Within this routine the array is redistributed according to the new distribution stored in the temporary layout descriptor.
4. For each array `A` that is aligned with an array that is the object of a redistribute statement a call to the initialization routine `vhpf_init_rt_dsc` with the alignment information as argument is generated followed by a call to `vhpf_redist`.

When an array is redistributed free memory is allocated according to the new distribution of the array, followed by a physical transfer of data to the new location (if necessary), and finally the memory used to store the array with the old distribution is deallocated and the original layout descriptor is updated to reflect the change of distribution. All arrays that are aligned with an array that is redistributed also may have to be redistributed in order to maintain the alignment relationship. Note that the runtime system routine `vhpf_redist` does not perform any data transfers if an array that is being redistributed does not yet have a distribution.

The transformation of `REDISTRIBUTE` statements is shown in Example 4.4.

Remarks

It should be noted that dynamic memory allocation and dynamic data redistribution are orthogonal issues. Therefore, an array may have the `DYNAMIC` attribute regardless of its allocation class. Similarly, an array may have the `ALLOCATABLE` or `POINTER` attribute and a static or a dynamic distribution. If, for example, an array is both dynamically allocated and dynamically distributed then a call to the initialization routine is generated whenever the shape or the distribution of the array is changed.

Currently a dynamically distributed array is declared in the target program as a *deferred shape array* with the `POINTER` attribute regardless of its original storage class. The main reason for this

[5] This is an array that has the `DYNAMIC` attribute.

Example 4.4: Transformation of Dynamically Distributed Arrays

original declaration of dynamically distributed arrays and `REDISTRIBUTE` *statement ...*

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

!HPF$ PROCESSORS R(NUMBER_OF_PROCESSORS())
!HPF$ DYNAMIC                        :: A
!HPF$ DYNAMIC, ALIGN (I,*) WITH A(I) :: B

      ...  ! first statement

!HPF$ REDISTRIBUTE(BLOCK) ONTO R :: A
```

generated Fortran 90 code ...

```
      REAL, POINTER, DIMENSION(:)   :: a
      REAL, POINTER, DIMENSION(:,:) :: b
      TYPE (rt_dsc), POINTER        :: a_dsc
      TYPE (rt_dsc), POINTER        :: b_dsc
      TYPE (rt_dsc), POINTER        :: a_dsc_tmp  ! temporary layout descriptor
      TYPE (rt_dsc), POINTER        :: b_dsc_tmp  ! temporary layout descriptor

      ! initialize descriptor of A with shape info only
      CALL vhpf_init_rt_dsc(dlist, a_dsc, 2, shape=(/N/), proc=r_dsc)

      ! initialize descriptor of B with shape info only
      CALL vhpf_init_rt_dsc(dlist, b_dsc, 2, shape=(/N,M/), proc=r_dsc)

      ...  ! first statement

      ! initialize temporary descriptor of A with new distribution info
      CALL vhpf_init_rt_dsc(dlist, a_dsc_tmp, 2, dtype=(/BLOCK/), proc=r_dsc)

      ! call redistribution routine for A
      CALL vhpf_redist(dlist, a, a_dsc, a_dsc_tmp)

      ! initialize temporary descriptor of B with alignment info
      CALL vhpf_init_rt_dsc(dlist, b_dsc_tmp, 2, alseq=(/1,0/), altrg=a_dsc)

      ! call redistribution routine for B
      CALL vhpf_redist(dlist, b, b_dsc, b_dsc_tmp)
```

This example shows the transformation of `REDISTRIBUTE` *statements. Prior to each* `REDISTRIBUTE` *statement a call to initialize a temporary descriptor with the new distribution is generated. Then a call to the runtime system routine* `vhpf_redist` *to perform the actual redistribution is inserted. Within this routine the new distribution of* `C` *is instantiated taking the shape information from the original descriptor and the distribution information from the temporary descriptor. Finally, the temporary descriptor is deallocated. Similar calls are generated for the alignee* `B` *in order to maintain the alignment relationship between* `B` *and* `A`*.*

strategy is to alleviate the handling of dynamic redistribution which usually requires to reallocate the memory within a runtime system procedure whenever a redistribution is performed.

However, experiments have shown that with some Fortran 90 compilers (e.g. SUN F90) the use of the POINTER attribute may result in a performance penalty for accessing such arrays. One reason for this may be that in the absence of appropriate analysis information a compiler cannot assume that an array with the POINTER attribute occupies contiguous storage. As a consequence, certain optimizations during address calculation may not be performed.

With the current version of Fortran, the reallocation of a dummy array is only possible if it has the POINTER attribute. Dummy arrays must not be given the ALLOCATABLE attribute. This weakness of Fortran is expected to be removed in Fortran 2000.

4.6 Transformation of Function Calls

Calls to *array intrinsic functions* with distributed arguments are substituted by calls to corresponding versions of these functions from the runtime system library as shown below. Currently only certain array intrinsic functions may be used for distributed arrays, including SIZE, SHAPE, LBOUND, and UBOUND.

Example 4.5: Substitution of Function Calls

```
!HPF$ DISTRIBUTE(BLOCK) :: D
      ...
 s1   D(1) = SIZE(D)                         ! array intrinsic function

 s2   D(I) = FUNCTION_WITH_SIDE_EFFECT(I)    ! user-defined function
```

transformed (intermediate) code ...

```
 s1   d(1) = vhpf_size(d_dsc)

 s2   d_tmp = function_with_side_effect(i)
      d(i) = d_tmp
```

Intrinsic functions called with distributed arguments are substituted by corresponding functions from the runtime system library which take layout descriptors as arguments. User-defined functions which may have side effects are substituted to enforce that they are executed on each processor.

Whenever a *user-defined function* that may cause side effects is referenced within an assignment statement that updates a distributed array, it is necessary to substitute the function call by a temporary variable and insert a corresponding assignment to this temporary variable prior to the original assignment statement as shown in Example 4.5. This transformation is required since assignments that update distributed arrays are transformed by VFC such that they are executed according to the owner computes rule only by the processor(s) owning the left-hand-side. In order to guarantee that the semantics are not changed in the presence of side effects, it is necessary to enforce that *all* processors execute the function call. Note that this transformation is not performed for PURE functions or intrinsic functions.

4.7 Transformation of Procedure Calls and Procedure Interfaces

VFC analyzes all calls to procedures and determines whether the distribution of an actual argument and the corresponding dummy argument are different, or if a distributed actual argument is associated with a non-distributed actual argument or vice versa. This analysis utilizes information from interface blocks (if present) with the goal to avoid implicit redistributions or copy-in/copy-out argument transfers whenever possible.

Example 4.6: Transformation of Procedure Calls and Interfaces

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

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

      INTERFACE
         SUBROUTINE SUB(X,Y)
!HPF$       PROCESSORS Q(NUMBER_OF_PROCESSORS())
!HPF$       DISTRIBUTE(BLOCK) ONTO Q :: X
            REAL, DIMENSION(:) :: X,Y
         END SUBROUTINE SUB
      END INTERFACE
      ...

      CALL  SUB(A,B)
```

generated code ...

```
      ...

      INTERFACE
         SUBROUTINE sub(x, y, x_dsc)               ! transformed interface
!HPF$       PROCESSORS q(1:number_of_processors())
            REAL, POINTER :: x(:)
!HPF$       DISTRIBUTE (BLOCK) ONTO q :: x
            REAL y(:)
            TYPE (rt_dsc), POINTER :: x_dsc
         END SUBROUTINE sub
      END INTERFACE
      ...

      CALL sub(a, b, a_dsc)                        ! pass descriptor of A
```

This example shows the handling of procedure calls and interfaces in the context of distributed arguments. Note that the interface block is required since the external `SUB` *contains distributed dummy arguments. The call statement and the interface are transformed by passing in addition to a distributed argument its layout descriptor.*

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*.

If a dummy array is explicitly distributed (e.g. 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.

If a dummy argument has the `DYNAMIC` attribute, the actual array must also have the `DYNAMIC` attribute and may have any *permissible* distribution.

The interface of a procedure with distributed arguments[6] is transformed such that for every distributed argument an additional dummy argument is generated in order to pass the runtime layout descriptor.

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 as shown in Example 4.7. If such a redistribution is generated for a procedure call, or if arguments have to be copied, the user is informed about this potential runtime overhead by corresponding *diagnostic messages*.

Example 4.7: Redistribution at Procedure Boundaries

original code ...

```
!HPF$ DISTRIBUTE(BLOCK) ONTO R :: A
      REAL, DIMENSION(N) :: A
      ...
      INTERFACE
         SUBROUTINE SUB(X)
            REAL, DIMENSION(:) :: X      ! X is not distributed
         END SUBROUTINE SUB
      END INTERFACE
      ...

      CALL  SUB(A)
      ...
```

generated code ...

```
      ...

      CALL vhpf_undist(a, dlist, a_dsc)    ! replicate array
      CALL sub(a)
      CALL vhpf_restore(a, dlist, a_dsc)   ! restore original distribution
      ...
```

In this example a distributed actual array `A` *is passed to a non-distributed formal array* `X`*. This is handled by replicating the array before the call to* `SUB` *by means of the runtime system routine* `vhpf_undist` *and by restoring the original distribution after the call by means of the runtime system routine* `vhpf_restore`*. Note, that VFC informs the user of the potential runtime overhead caused by these redistributions by issuing a corresponding* diagnostic message*. Note that the interface of* `SUB` *is required since it has an assumed-shape dummy argument.*

[6] VFC supports dummy arguments that have the `DISTRIBUTE, INHERIT,`, `DYNAMIC`, or `RANGE` attribute.

4.8 Parallelization of Array Assignments

Array assignments are parallelized based on the *owner computes* paradigm according to the ownership of the left-hand-side array section. Unlike most HPF compilers, array statements are not converted into loops, but are transformed to array statements that operate on the local memory on each processor. The main advantage of this strategy is that the parallelism of the original construct is preserved. Therefore, both inter- and intra-processor parallelism can be exploited by the Fortran 90 compiler of the parallel target machine.

Before transforming array assignments which involve distributed objects, VFC analyzes the code to determine the parallelization strategy that should be applied. Currently three different classes of array assignments that involve distributed arrays are distinguished and handled with varying overheads.

- **Class 1 Array Assignments**
  The first class of array assignments can be handled in an optimal way without the need for any transformations. This requires that for all arrays on the right-hand-side each dimension with a non-scalar subscript is distributed in exactly the same way as the corresponding dimension of the left-hand-side array and that all corresponding distributed dimensions of all arrays involved are accessed with identical section subscripts and as a whole.

- **Class 2 Array Assignments**
  The second class comprises array assignment statements for which only work distribution and index conversion has to be performed, but which can be processed without introducing inter-processor communication. This strategy is applied if, for all arrays on the right-hand-side, each dimension with a non-scalar subscript is distributed in exactly the same way as the associated dimension of the left-hand-side array. Such statements are transformed by adapting the section-subscripts of distributed dimensions in such a way that each processor only accesses its local data. These local section-subscripts are determined at runtime by intersection of the local index set of an array dimension with the original global section subscripts and converting the resulting section into local indices.

- **Class 3 Array Assignments**
  All other array assignment statements are transformed by introducing appropriate calls to run-time library routines that determine the work distribution and generate the required communication of non-local data into temporary arrays.

4.8.1 Sketch of Classification Algorithm

Assume an array assignment of the following form:

`A(`*subscript-list-A*`) = B(`*subscript-list-B*`)`

Let r denote the rank of the left-hand-side array `A` and q the rank of the right-hand-side array `B`, respectively. The *left-to-right sequence* of `A` with respect to `B`, is a list of r integer values $ltr_1, \ldots ltr_r$, where each ltr_i, $1 \leq i \leq r$, is

- either the value $d, 1 \leq d \leq q$, iff the subscript in dimension i of `A` is a section subscript and dimension d of `B` corresponds to dimension i of `A`, or

- 0, otherwise.

For the above array assignment

1. **no communication is required**, iff $\forall i, 1 \leq i \leq r \wedge ltr(i) \neq 0$, the following two conditions are satisfied:

   (a) The distribution of dimension i of `A` is the same as of dimension $ltr(i)$ of `B`.

   (b) The *section-subscript* of dimension i of the left-hand-side array section is the same as of dimension $ltr(i)$ of the right-hand-side array section.

2. **no workdistribution is required**, iff

   (a) no communication is required (see 1), and

   (b) all accesses in distributed dimensions of `A` and `B` are subject to the whole dimension.

If an array assignment references more than one array section on the right-hand-side, then the above conditions must be satisfied for each array section on the right-hand-side.

Prior to the classification as outlined above, array assignments are normalized by substituting all references to distributed arrays that occur within a *subscript* with temporary variables. Depending on their class, array assignment statements are transformed as follows:

For array assignments of class 1 each *section-subscript* of a distributed dimension is transformed such that on each processor only the local section is referenced. Since in an array assignment of class 1 all distributed dimensions have to be accessed as a whole, no conversion of global to local indices is required.

Array assignments of class 2 are transformed in several separate steps. First for each access to a distributed dimension of the array section on the left-hand-side a call to the work distribution routine **`vhpf_work_dist`** is called with the layout descriptor, the number of the accessed dimension, and the corresponding section information as argument. Within this routine the section that has to be computed by the executing processor is determined by intersection of the local index set in the corresponding array dimension on this processor with the referenced global section. The result of this intersection is converted to local indices and returned. Depending on the distribution of the accessed array dimension and the nature of the access (i.e. section subscript or vector subscript) the result of this intersection is either a single triplet, a set of triplets or an array of indices. The array assignment is thus transformed by substituting the original subscripts of the mapped dimensions with the information returned by the work distribution function.

Array assignment statements of class 3 are usually substituted by a call to the runtime library routine **`vhpf_sec_assign`** which is capable of assigning an arbitrary section of a distributed array to an arbitrary section of another distributed array. The details of communication generation, work distribution and index conversion are hidden within this routine. Prior to the call to `vhpf_sec_assign` so-called *section descriptors* are generated in order to pass the information about the accessed array sections to the runtime library routine. If multiple array sections are referenced on the right-hand-side, the array assignment statement is split in such a way that for each array section `B(`*subscript-list-B*`)` on the right-hand-side a temporary array `B_TMP` with the same shape and distribution as the left-hand-side array `A` is generated, and an assignment, `B_TMP(`*subscript-list-A*`) = B(`*subscript-list-B*`)`, of the original array section to the temporary array is generated in front of the original statement. This newly introduced array assignment statement is then transformed as described above (class 2). The section `B(`*subscript-list-B*`)` on the right-hand-side is then replaced by the section `B_TMP(`*subscript-list-A*`)` and the resulting array assignment is transformed according to the transformations of class 2.

Example 4.8: Transformation of Array Assignments

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

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

! class 1 array assignment
A(:,:) = B(:,I,:)

! class 2 array assignment
A(1:M:2,:) = B(1:M:2,I,:)

! class 3 array assignment
A(:L,:) = C(:,:)
```

transformed code ...

```
! class 1 array assignment
a(1:vhpf_loc_ub(a_dsc, 1):1, 1:vhpf_loc_ub(a_dsc, 2):1) = &
    b(1:vhpf_loc_ub(b_dsc, 1):1, i, 1:vhpf_loc_ub(b_dsc, 3):1)

! class 2 array assignment
CALL vhpf_work_dist(a_dsc, dim=1, l=1, u=m, s=2, ll=ll_1_tmp, lu=lu_1_tmp, ls=ls_1_tmp)
CALL vhpf_work_dist(a_dsc, dim=2, l=1, u=n, s=1, ll=ll_2_tmp, lu=lu_2_tmp, ls=ls_2_tmp)
a(ll_1_tmp:lu_1_tmp:ls_1_tmp, ll_2_tmp:lu_2_tmp:ls_2_tmp) = &
  b(ll_1_tmp:lu_1_tmp:ls_1_tmp, i, ll_2_tmp:lu_2_tmp:ls_2_tmp)

! class 3 array assignment
CALL vhpf_set_secdsc(a_sec, rank=2, l1=1, u1=u1, l2=1, u2=n)
CALL vhpf_set_secdsc(c_sec, rank=2, l1=1, u1=u2, s1=2, l2=1, u2=n)
CALL vhpf_sec_assign(a, a_dsc, a_sec, c, c_dsc, c_sec)
```

This example shows the transformation of array assignments. The first array assignment statement does neither require work distribution nor communication. The runtime system routine `vhpf_loc_ub` *returns the upper bound of the local segment on the executing processor excluding potential buffer space.*

The second array assignment statement requires work distribution but no communication. The runtime system routine `vhpf_work_dist` *takes besides the layout descriptors, the information about the accessed array section in the corresponding dimension as arguments and returns the information about the section that has to be computed by the calling processor in local indices. This information is then used to transform the array assignment such that on each processor the corresponding local array section is accessed.*

The third array assignment requires both work distribution and communication, and is transformed into a call to the runtime system routine `vhpf_sec_assign` *which can handle the assignments of arbitrary sections of distributed arrays. The details of work distribution, index conversion, and buffer management for non-local data are hidden inside of this procedure.*

4.9 Parallelization of Independent Loops

The compilation techniques applied for parallelizing independent loops differ significantly from those applied to other parts of the program[7]. Since in such loops arrays are usually accessed indirectly (via indirection arrays defined at runtime) compile-time analysis techniques cannot be applied with reasonable efficiency. Therefore, the transformation of independent loops combines compile time parallelization techniques with runtime analysis, based on the inspector/executor paradigm. These techniques utilize the fact that there are no loop-carried dependences in the loop, and thus the execution of an iteration cannot change data items used by any other iteration. Consequently, any communication required for accessing data items owned by other processors can be performed before the loop, and similarly, non-local data items assigned to by an iteration can be communicated back to the processors that own them at the end of the loop. Runtime analysis and parallel execution is supported by VFC's runtime library as described in Section 5.

The parallelization strategy of VFC for independent loops has been fully redesigned and generalized. Opposed to VFCS, the generalized strategy is also applicable to non-perfectly nested loops and arrays with arbitrary multidimensional distributions.

Communication schedule reuse is supported based on the HPF+ conditional `REUSE` clause which constitutes an assertion to the compiler that all communication schedules remain either invariant for all executions of an `INDEPENDENT` loop, or that they change only if the condition associated with the reuse clause has been changed. This implies that the inspector needs to be executed only once, or in case of a conditional reuse clause, whenever the reuse-condition evaluates to *false*.

The `REDUCTION` clause and *reduction statements* are provided in order to specify reduction operations.

The `ON HOME` clause can be used to specify how the iterations of an `INDEPENDENT` loop should be distributed to the processors. In addition to the `ON HOME` clause, the HPF+ `RESIDENT` clause allows the user to assert to the compiler that only local array elements are accessed in the loop. As a consequence of this no communication needs to be generated.

Moreover, if for a distributed array used inside the independent loop the *halo* has been specified then its transformation can be simplified significantly.

Despite these major advances there are still some restrictions on independent loops. If any of theses current restrictions is violated the iteration space of the loop is not distributed or, in the worst case, the loop is transformed based on the transformation strategy of scalar assignments as described in Section 4.10. If this is the case, the user is informed by corresponding *diagnostic messages*. The current restrictions are summarized in the HPF+ User's Guide [7].

The remainder of Section 4.9 is organized as follows. In Sections 4.9.1-4.9.7 we outline the general parallelization strategy for independent loops, and in Section 4.9.8 several optimization techniques are described.

[7]Note that currently, all `INDEPENDENT` loops are parallelized with the inspector/executor strategy.

4.9.1 Compilation Scheme

Independent loop nests are processed as a monolithic entity, i.e. the inspector/executor parallelization strategy is applied to the whole loop nest. The transformation of an independent loop nest can be described as a sequence of several separate phases, as outlined in the following:

- *Construction and initialization of the loop nest descriptor*
- *Generation of auxiliary data structures*
- *Handling of reduction operations*
- *Generation of work distributor*
- *Generation of inspectors*
- *Generation of calls to communication routines*
- *Transformation of loop nest body*

The first two steps can be described as the *analysis phase* and the next ones as *code generation phases*. The analysis phase traverses the program, analyzes the structure of the independent loop nest, and constructs internal data structures. The code generation phases perform the actual code transformation by modifying the abstract syntax tree. This clean separation enhances the modularity of the compiler and alleviates the development and debugging processes.

Generally, an independent loop nest is transformed into three main code segments: *work distributor*, *inspector*, and *executor*. Furthermore, for each independent loop nest several auxiliary data structures are declared and dynamically maintained, which are utilized in the generated code.

On each processor, the work distributor computes the execution space, i.e. set of loop iterations, to be executed on this processor. The inspector performs the runtime analysis of the independent loop nest, and for every distributed array generates descriptions of the communication patterns in the form of *schedules* utilized by communication procedures of the subsequent executor phase. Note that communication may be required before the executor loop and after the executor loop if a strategy other than *owner computes* has been used. Runtime analysis and communication may be omitted for some special cases (see Section 4.9.8).

4.9.2 Loop Nest Descriptor

In the first step high-level information structures are constructed which serve as a framework for the target code generation. Several compile-time optimizations are performed with the goal of eliminating redundant runtime preprocessing.

The compile-time information about a particular independent loop nest is represented by means of *loop nest descriptors*, which is a list structure where each loop inside the independent loop nest is condensed to a single item in the list, as shown in Figure 1.

Associated with each independent loop nest is a list of *compile-time array descriptors* for all arrays occurring in the loop nest. Every compile-time array descriptor summarizes information about characteristics of the array and records all its references, as shown in Figure 2.

```
typedef struct doind_node {          /* loop descriptor                      */
    struct doind_node  *next;        /* next included loop                   */
    tTree              loop_ref;     /* pointer to loop statement in AST     */
    int                seq_num;      /* sequence number                      */
    int                seqpar;       /* PARLOOP :  independent loop          *
                                      * SEQLOOP :  sequential loop           */
    PTypeIdentEntry    loop_entry;   /* pointer to loop iteration var        */
    tTree              loop_init;    /* pointer to loop init bound in AST    */
    tTree              loop_final;   /* pointer to loop final bound in AST */
} DOIND_NODE;
```

Figure 1: Loop nest descriptor

The loop nest descriptor of an independent loop is constructed and partially initialized by traversing the abstract syntax tree of the loop nest. These initial settings are updated in subsequent analysis phases. For each array reference an *offset number* and a *correspondence vector* is calculated.

- The offset number represents the position of the array element in the local index space. All references of an array are symbolically compared with each other and, if they are equal, they are associated with the same offset number. Offset numbers are calculated for every indirection level separately.

- The correspondence vector determines for each dimension of a particular array reference those enclosing loops (if any) whose iteration variables are referenced in the corresponding subscript. The correspondence vector is represented as n-tuple of the constants CV_CONST, CV_SCALAR, and CV_LOOP, where n is the rank of the array. In the case that the subscript references loop iteration variables – correspondence value is equal to CV_LOOP, and a list recording the *sequence numbers* of the corresponding loops is associated to it. Correspondence vectors are used to generate the inspector phase and to collect global array indices.

For every distributed array occurring inside the independent loop nest the access patterns are analyzed in order to determine the transformation strategy that should be applied. First each array reference is symbolically compared with the array variable referenced in the on-clause in order to detect local array reference. This comparison is performed separately for each dimension of an array reference. If all references of an array are found to be local, no calls to communication procedures need to be generated. Depending on whether a subscript in a particular array dimension is local, and on the distribution type and the form of the subscript in this dimension different transformation schemes are chosen for it. For more details see Section 4.9.8.

```
typedef struct arr_occur {          /* array reference descriptor        */
    struct arr_occur *next;         /* next reference                    */
    tTree            aref;          /* pointer to array reference in AST  */
    tTree            stmt;          /* pointer to enclosing stat. in AST  */
    int              offset_num;    /* offset number                     */
    int              ind_level;     /* indirection level                 */
    int              diff;          /* live/dead reference               */
    int              cvector[7];    /* correspondence vector:            *
                                     * CV_ERR    : error                 *
                                     * CV_INIT   : initial value         *
                                     * CV_CONST  : constant              *
                                     * CV_SCALAR : scalar                *
                                     * CV_LOOP : includes loop iter vars *
                                     */
    struct cv_do    *cv_dolist[7];  /* pointers to list of loops         */
} ARR_OCCUR;

typedef struct doind_arr_node {     /* array descriptor                  */
    struct doind_arr_node *next;    /* next array                        */
    PTypeIdentEntry   arrident;     /* pointer to array in symbol table  */
    struct arr_occur  *def_occ;     /* pointer to 'def' arr_occur list   */
    struct arr_occur  *use_occ;     /* pointer to 'use' arr_occur list   */
    int               *def_refnum;  /* number of different 'def' refs    */
    int               *use_refnum;  /* number of different 'use' refs    */
    int               mapped;       /* mapping attribute                 */
    int               usedef;       /* 'use' or/and 'def' attribute      */
    int               multi_lev;    /* accesses on multi levels          */
    int               max_lev;      /* max indirection level of accesses */
    int               multi_cv;     /* multi loop dependence             */
    int               reduct_op;    /* reduction operator                */
    int               inif;         /* array occurs in IF-statement      */
    int               use_eq_def;   /* 'use' refs are equal to 'def' refs */
    int               def_transf[3]; /* transformation strategy for 'def' */
    int               use_transf[3]; /* transformation strategy for 'use' */
                                    /* NOTR        : no transformation   *
                                     * TR_INSPEX   : via DVIEW()         *
                                     * TR_RESIDENT : via AGGREGATE_G2L() *
                                     * TR_LOCAL    : via AGGREGATE_G2L() *
                                     * TR_SIMPLE   : i => local_iter     *
                                     * TR_LINEAR   : n*i+d =>            *
                                     *    n*(local_iter-1)+start_index   *
                                     */
} DOIND_ARR_NODE;
```

Figure 2: Compile-time array descriptor

4.9.3 Auxiliary Data Structures

Runtime analysis and parallel execution requires the compiler to generate data structures to store results computed during inspection. For example, variables for managing local iteration spaces, array subscript lists, schedules, etc. are generated. The names of the new variables are closely related to the names of arrays for which they are generated, and they are reused as much as possible. Some of the generated data structures are shown in Figure 3.

```
TYPE ITER_BOX
  INTEGER, POINTER   :: ITERS(:)     ! iterations
  INTEGER            :: LEN          ! number of iterations
END TYPE ITER_BOX

TYPE ROW
  INTEGER, POINTER   :: INDX(:)      ! subscript list
END TYPE ROW

TYPE INDX_BOX
  TYPE(ROW), POINTER :: DIM(:)       ! dimensional subscript list
  INTEGER, POINTER   :: OFFSET(:,:)  ! dimensional offset sets
  INTEGER            :: N            ! rank of the array
  INTEGER            :: NOCCUR       ! number of references
END TYPE INDX_BOX

TYPE (ITER_BOX), POINTER :: EXEC(:)

TYPE (INDX_BOX)  :: X_GBOX
TYPE (INDX_BOX)  :: X_LBOX

INTEGER, POINTER :: X_OFFSET(:,:)
INTEGER, POINTER :: X_OFFPROC(:)

INTEGER          :: X_SCHED
```

Figure 3: Auxiliary Data Structures

The auxiliary data structure EXEC is used to represent the local iteration space of an independent loop nest on the executing processor, referred to as *execution space*. The size of EXEC is equal to the nesting depth n of the loop nest, i.e. EXEC(i) stores the execution space of the loop with nesting level $i-1$, $1 \leq i \leq n$. EXEC(1) always contains the iteration space for the outermost independent loop with respect to the executing processor. Local iterations to be performed on the executing processor, are stored in the component ITERS, and are determined at runtime by the work distributor.

For each distributed array X referenced in an independent loop the following auxiliary data structures are generated (see also Figure 3).

- `X_GBOX` and `X_LBOX` are used to store the dimensional subscript lists. Subscripts of all references of X in global name space are collected in `X_GBOX`, and the corresponding subscripts in local name space are placed into `X_LBOX`.
- `X_OFFSET` is used to store dimensional location counters for all array references.
- `X_OFFPROC` contains the number of non-local indices in every dimension.
- `X_SCHED` represents the communication schedule and is implemented as a pointer to a C data structure.
- For each distributed array X its runtime layout descriptor will be denoted by `X_DSC`.

Generally, the auxiliary data structures are allocated/deallocated dynamically for each independent loop nest by means of calls to corresponding runtime library routines.

```
             REAL, DIMENSION(N2,N1) :: a
             REAL, DIMENSION(N1,N2) :: b, d
             REAL, DIMENSION(N1)    :: c

             INTEGER, DIMENSION(N1) :: x
             INTEGER, DIMENSION(N2) :: y
             ...

       !HPF$  PROCESSORS :: PP1(NP*NP), PP2(NP,NP)=RESHAPE(PP1)
       !HPF$  DISTRIBUTE(BLOCK, BLOCK) ONTO PP2 :: a, b, d
       !HPF$  DISTRIBUTE(BLOCK) ONTO PP1        :: x
       !HPF$  DISTRIBUTE(BLOCK) ONTO PP1, HALO(halof(i)) ON PP1(i) :: c
             ...

       !HPF$  INDEPENDENT, ON HOME(a(:,i)), RESIDENT(d)
(L1)          DO i = 1, N1
       !HPF$    INDEPENDENT, ON HOME(a(j,i))
(L2)            DO j = 1, N2
                  a(j,i) = (b(i,j) + b(x(i),y(j))) * c(x(i)) + d(i,j)
                END DO
              END DO
```

Figure 4: Example of nested independent loops

4.9.4 Work Distributor

On each processor p, the work distributor computes the *execution space* for the loop nest, i.e. the set of loop iterations for every loop included inside the loop nest, to be executed on processor p. The current implementation requires that the iteration space of independent loop nests is rectilinear, and thus can be constructed by means of the Cartesian product of iterations spaces of the individual loops. Consequently, the iteration space of an inner loop must not dependent on the iteration space of an outer loop. The execution space is determined for each loop contained in the independent loop nest separately, referred to as *dimensional execution set*, according to the `ON HOME` or `ON` clause specified in the `INDEPENDENT` directive. If no on-clause is present, it is derived automatically by the compiler. In the case that the compiler fails in generating an on-clause automatically, the loop iteration space will not be distributed.

Work distribution is implemented by a call to a runtime library routines of the `VHPF_BUILD_EXEC_xxx` family of routines. These routines determine the execution set on the executing processor according to the ownership of a certain distributed array dimension. Which variant of these routines will be chosen depends on the distribution and structure of the subscript in the corresponding dimension of the array variable of the on-home clause. The dimensional execution set for a sequential loop is equal to the whole iteration set.

Consider the independent loop nest[8] of Figure 4, where `A, B, C, D`, and `X` are distributed arrays, and `Y` is replicated. The corresponding work distributor code together with auxiliary variable declarations is shown in Figure 5. The routine `VHPF_BUILD_EXEC_OWB` implements an optimized version of the work distribution algorithm which is utilized whenever the array used in on-home clause has a `BLOCK` or `GEN_BLOCK` distribution and the subscript is a linear function of the loop iteration variable.

```
! -<< Declarations of auxiliary variables >>-
      TYPE (ITER_BOX), POINTER :: exec(:)
      TYPE (INDX_BOX)  :: a_gbox, a_lbox, b_gbox, b_lbox, x_gbox, x_lbox
      INTEGER, POINTER :: a_offset(:,:), b_offset(:,:), x_offset(:,:)
      INTEGER, POINTER :: a_offproc(:), b_offproc(:), x_offproc(:)
      ...
      ALLOCATE(exec(1:2))
! -<< Dimensional work distribution for loop (L1) ==> EXEC(1) >>-
      CALL VHPF_BUILD_EXEC_OWB(a_dsc, 1, N1, 1, 0, 2, 1, exec)

! -<< Dimensional work distribution for loop (L2) ==> EXEC(2) >>-
      CALL VHPF_BUILD_EXEC_OWB(a_dsc, 1, N2, 1, 0, 1, 2, exec)

! -<< Allocations of auxiliary variables using information computed
!     in the analysis phase, and in the work distributor      >>-
       ...
```

Figure 5: Work distributor code

[8] The loop nest has been constructed for the purpose of illustrating the generated target code in the presence of arrays with different access patterns

4.9.5 Inspector

The inspector performs a runtime analysis of the independent loop nest which is outlined in the following.

The main task of the inspector is to generate a description of the communication necessary for the loop nest in the form of *schedules*, which are used to control runtime procedures for communicating data between processors in the subsequent executor phase.

Moreover, the inspector establishes an addressing scheme for converting global indices into local indices. Information needed to generate schedules, to allocate communication buffers, and for index conversion is obtained from the appropriate *global reference list* and the runtime layout descriptors. A *global reference list* for a distributed array X is constructed from its dimensional subscript functions, taking into account all references to the array.

Subscripts are collected individually by means of a do-loop iterating over the corresponding execution set and stored in `X_GBOX`. Symbolic analysis ensures that only textually distinct subscripts are included in the list. In the case of multi-level indirections, array accesses are processed in multi-phases, starting with the innermost arrays. This order guarantees that non-local references to distributed indirection arrays are resolved in previous phases.

Results of global to local index conversion are stored in the corresponding *local reference lists* of `X_LBOX`. Local reference lists are built in such a way that the local index refers either to the local section or to the buffer which is adjacent to the local section and contains the copies of non-local elements referenced.

Global to local index conversion and construction of communication schedules is performed within the runtime procedure `VHPF_DVIEW`. Besides computing the schedule, `VHPF_DVIEW` determines the total number of non-local indices, stored in `X_OFFPROC`, for every array dimension. If the predefined buffer area for copies of non-local elements is too small, the array can be re-allocated with a larger buffer[9].

If the compiler succeeds in detecting that only local elements of an array are accessed, or the array is referenced in the `RESIDENT` clause of the loop nest, no calls to communication procedure are generated and the runtime procedure `PARS_AGGREGATE_G2L` is utilized for an optimized global to local index conversion.

If an array has a halo, then a communication schedule is computed once, at the time the array is allocated. In this case, during the inspector phase only call to the runtime library procedure `PARS_HALO_INDXMAPR` is generated for performing the index conversion.

Consider the independent loop nest given in Figure 4. After the work distributor phase the calculated execution sets are saved in `EXEC(1)` for loop (L1), and `EXEC(2)` for loop (L2). The inspector code corresponding to the loop nest is shown in Figure 6. Array accesses are processed in multiple phases starting with the innermost subscripts. In each phase only array references with the same indirection level are resolved. For a distributed array `X` which is used as an indirection array of the distributed arrays `B` and `C`, all non-local elements must be made available to the executing processor by means of communication before the references to `B` and `C` can be resolved. Note that for array `A` no inspector is necessary, because the only reference to `A` is identical with the variable reference in the on-clause.

[9]Note, that *dynamic buffer management* must be enabled by the command-line option "-b"

```
! -<< Process global subscripts of X >>-
       offs = 0
       x_offset(1,1) = offs         ! 1st reference, 1st dimension
       DO ni = 1, exec(1)%len
         x_gbox%dim(1)%indx(ni+offs) = exec(1)%iters(ni)
       END DO
       offs = offs + exec(1)%len
       x_offset(1,2) = offs
       x_gbox%noccur =  1;        x_lbox%noccur = 1
       x_gbox%offset => x_offset; x_lbox%offset = x_offset
! -<< Global to local index conversion and construction of schedule for X >>-
       CALL VHPF_DVIEW(x_dsc, x_gbox, x_sched, x_lbox, x_offproc, 0, lseg_1d=x)
! -<< Gather non-local elements of X >>-
       CALL VHPF_IGATHER(x_dsc, x, x_sched)
! -<< Free schedule for X >>-
       CALL VHPF_FREE_SCHED(x_sched)

! -<< Process global subscripts of B >>-
       offs = 0
       b_offset(1,1) = offs         ! 1st reference, 1st dimension
       DO ni = 1, exec(1)%len
         b_gbox%dim(1)%indx(ni+offs) = exec(1)%iters(ni)
       END DO
       offs = offs + exec(1)%len    ! 2nd reference, 1st dimension
       b_offset(1,2) = offs
       DO ni = 1, exec(1)%len
         i = ni + x_lbox%offset(1,1)
         b_gbox%dim(1)%indx(ni+offs) = x(x_lbox%dim(1)%indx(i))
       END DO
       offs = offs + exec(1)%len
       b_offset(1,3) = offs
       offs = 0
       b_offset(2,1) = offs         ! 1st reference, 2nd dimension
       DO ni = 1, exec(2)%len
         b_gbox%dim(2)%indx(ni+offs) = exec(2)%iters(ni)
       END DO
       offs = offs + exec(2)%len
       b_offset(2,2) = offs         ! 2nd reference, 2nd dimension
       DO ni = 1, exec(2)%len
         j = exec(2)%iters(ni)
         b_gbox%dim(2)%indx(ni+offs) = y(j)
       END DO
       offs = offs + exec(2)%len
```

Figure 6: Inspector code (part 1)

```
       b_offset(2,3) = offs
       b_gbox%noccur =  2;        b_lbox%noccur = 2
       b_gbox%offset => b_offset; b_lbox%offset = b_offset

! -<< Global to local index conversion and construction of schedule for B >>-
       CALL VHPF_DVIEW(b_dsc, b_gbox, b_sched, b_lbox, b_offproc, 0, lseg_2d=b)

! -<< Process global subscripts of C >>-
       offs = 0
       c_offset(1,1) = offs         ! 1st reference, 1st dimension
       DO ni = 1, exec(1)%len
         i = ni + x_lbox%offset(1,1)
         c_gbox%dim(1)%indx(ni+offs) = x(x_lbox%dim(1)%indx(i))
       END DO
       offs = offs + exec(1)%len
       c_offset(1,2) = offs
       c_gbox%noccur =  1;        c_lbox%noccur = 1
       c_gbox%offset => c_offset; c_lbox%offset = c_offset

! -<< Global to local index conversion using halo region specification >>-
       CALL PARS_HALO_INDXMAPR(c_dsc, c_gbox, c_lbox)

! -<< Process global subscripts of D >>-
       offs = 0
       d_offset(1,1) = offs         ! 1st reference, 1st dimension
       DO ni = 1, exec(1)%len
         d_gbox%dim(1)%indx(ni+offs) = exec(1)%iters(ni)
       END DO
       offs = offs + exec(1)%len
       d_offset(1,2) = offs
       offs = 0
       d_offset(2,1) = offs         ! 1st reference, 2nd dimension
       DO ni = 1, exec(2)%len
         d_gbox%dim(2)%indx(ni+offs) = exec(2)%iters(ni)
       END DO
       offs = offs + exec(2)%len
       d_offset(2,2) = offs
       d_gbox%noccur =  1;        d_lbox%noccur = 1
       d_gbox%offset => d_offset; d_lbox%offset = d_offset

! -<< Global to local index conversion; no communication is needed >>-
       CALL PARS_AGGREGATE_G2L(d_dsc, d_gbox, d_lbox)
```

Figure 6: Inspector code (part 2)

4.9.6 Executor

The executor is the final phase in the execution of the independent loop nest. It performs communication described by schedules (gather/scatter non-local array elements used/updated in the loop), and executes the loop nest body for the computed execution set.

Copies of non-local elements are placed in the buffer area which is appended immediately to the local segment of the array, for every dimension. All references to distributed arrays are transformed such that each global subscript of a distributed dimension is replaced by a local reference obtained from the local reference list which ensures that the correct data in local segments or buffers are accessed.

The executor code for the loop nest of Figure 4 is shown in Figure 7. For moving data between processors the library procedures `VHPF_GATHER` and `VHPF_SCATTER` are utilized. It is allowed to insert `IF` constructs into the independent loops. Then each array reference updated inside the `IF` is marked using the temporary mask array utilized when scattering non-local elements through the library procedure `VHPF_MASKED_SCATTER`. To increase the performance of the generated executor loop auxiliary data structures are used to store invariant variables in addressing array elements.

```
! -<< set up auxiliary variables for accessing arrays >>-
       b_tindx1 => b_lbox%dim(1)%indx;   b_tindx2 => b_lbox%dim(2)%indx
       b_toffs1_01 = b_lbox%offset(1,1); b_toffs2_01 = b_lbox%offset(2,1)
       b_toffs1_02 = b_lbox%offset(1,2); b_toffs2_02 = b_lbox%offset(2,2)
       c_tindx1 => c_lbox%dim(1)%indx
       c_toffs1_01 = c_lbox%offset(1,1)
       d_tindx1 => d_lbox%dim(1)%indx;   d_tindx2 => d_lbox%dim(2)%indx
       d_toffs1_01 = d_lbox%offset(1,1); d_toffs2_01 = d_lbox%offset(2,1)

! -<< Gather non-local elements of B >>-
       CALL VHPF_RGATHER(b_dsc, b, b_sched)

! -<< Free schedule for B >>-
       CALL VHPF_FREE_SCHED(b_sched)

! -<< Transformed loop nest body >>-
!HPF$  INDEPENDENT, ON HOME (a(1:n2, i))
       DO i = 1, exec(1)%len
!HPF$    INDEPENDENT, ON HOME (a(j, 1:n1))
         DO j = 1, exec(2)%len
           a(j, i) =                                                   &
              (b(b_tindx1(i + b_toffs1_01), b_tindx2(j + b_toffs2_01)) + &
               b(b_tindx1(i + b_toffs1_02), b_tindx2(j + b_toffs2_02)))* &
               c(c_tindx_1(i + c_toffs_1_01)) +                        &
               b(d_tindx1(i + d_toffs1_01), d_tindx2(j + d_toffs2_01))
         END DO
       END DO
```

Figure 7: Executor code

4.9.7 Reduction Operations

Independent loops must not have any loop-carried dependences and thus no scalar variable can be modified. This can only be achieved by using the `NEW` clause to define *private* variables that are conceptually local in each iteration or the `REDUCTION` clause which specifies variables occurring in reduction statements only. Both clauses are supported by VFC.

Reduction statements using the operators $+$, $-$, $*$, and $/$, and the reduction functions `MIN` and `MAX` have been implemented.

Reduction statements are processed in three steps:

1. Initialization of the scalar reduction variable or, if the reduction variable is a distributed array, of the array's buffer space.
2. Computation of the *local reduction* for the local iteration space of the transformed independent loop nest. For non-local elements of a distributed array the reduction process will take place in the buffer, without gathering the non-local elements before.
3. Computation of the *global reduction* by accumulating the local reductions of each processor after execution of the independent loop nest.

Consider the independent loop nest shown in Figure 8 which contains reduction statements for the scalar variable R and for the distributed array A, respectively. The executor code together with the initialization and accumulation phases is shown in Figure 9.

```
        REAL, DIMENSION(N1)    :: a, b
        INTEGER, DIMENSION(N1) :: x
        REAL                   :: r
        ...

!HPF$   PROCESSORS :: PP1(NP)
!HPF$   DISTRIBUTE(BLOCK) ONTO PP1 :: a, b, x
        ...

!HPF$   INDEPENDENT, ON HOME(x(i)), REDUCTION(a, r)
        DO i= 1, N1
          a(x(i)) = a(x(i)) * 10
          r = r + b(x(i))
        END DO
```

Figure 8: Example of independent loop with REDUCTION clause

```
! -<< set up auxiliary variables for accessing arrays >>-

       a_tindx1 => a_lbox%dim(1)%indx
       a_toffs1_01 = a_lbox%offset(1,1)
       b_tindx1 => b_lbox%dim(1)%indx
       b_toffs1_01 = b_lbox%offset(1,1)


! -<< Initialization >>-

       s = r
       r = 0.
       CALL VHPF_RSET_BUF(a_dsc, 1, lseg_1d=a)


! -<< Gather non-local elements of B >>-

       CALL VHPF_RGATHER(b_dsc, b, b_sched)
       CALL VHPF_FREE_SCHED(b_sched)


! -<< Transformed loop nest body >>-

!HPF$   INDEPENDENT, REDUCTION (a, r), ON HOME (x(i))
         DO i = 1, exec(1)%len
           a(a_tindx1(i + a_toffs1_01)) = a(a_tindx1(i + a_toffs1_01)) * 10
           r = r + b(b_tindx1(i + b_toffs1_01))
         END DO


! -<< Multiply non-local elements of A by their original values >>-
       CALL VHPF_RSCATTER_MUL(a_dsc, a, a_sched)
       CALL VHPF_FREE_SCHED(a_sched)


! -<< Calculate the global sum of r among all processors >>-
       CALL VHPF_RED(r, 1)
       r = r + s
```

Figure 9: Executor code with reduction operations

4.9.8 Optimizations

The goal of the optimization phase is to reduce the cost of runtime preprocessing performed in the inspector, which usually introduces a large overhead both in terms of execution time and memory.

4.9.8.1 Communication Schedule Reuse

The cost of executing an inspector can be amortized when the information produced by the inspector is computed only once and then used repeatedly. However, in many cases, the compiler does not have sufficient information to decide whether an inspector computation is redundant. Therefore, eliminating redundant inspectors is performed based on the `REUSE` clause which constitutes an assertion to the compiler that the communication schedule of all arrays accessed in the loop is either invariant for all executions of the loop, or it changes only if the condition associated with the reuse clause has been changed. This implies that inspector needs to be executed only once, or only in those cases where the reuse-condition evaluates to `.FALSE.`.

```
      !HPF$  INDEPENDENT, REUSE
(L1)         DO i= l1, u1
               ...
             END DO


      !HPF$  INDEPENDENT, REUSE(rcond)
(L2)         DO j= l2, u2
               ...
             END DO
```

Figure 10: Example of independent loops with REUSE clause

If the `REUSE` clause is present the work distributor and inspector are guarded by means of conditional statements to enforce that they are executed only if the reuse-condition is true or if the loop is executed for the first time. This is shown in Figure 11 for the independent loops of Figure 10.

It should be mentioned that the implementation of the `REUSE` clause may significantly increase the memory requirements of the parallelized program. Generally, all independent loops in the program unit without a reuse clause share the auxiliary data structures which are allocated before and deallocated after the loop. An independent loop with reuse clause must have its own auxiliary data structures which need to be saved for the next execution.

```
        LOGICAL, SAVE :: lfirst1 = .TRUE., lfirst2 = .TRUE.
        ...

!       loop (L1)
!
        IF (lfirst1) THEN
           -<< Work distributor (L1) >>-
           -<< Inspector (L1) >>-
          lfirst1 = .FALSE.
        END IF
        -<< Executor (L1) >>-


!       loop l2
!
        IF (lfirst2 .OR. .NOT.rcond) THEN
           -<< Work distributor (L2) >>-
           -<< Inspector (L2) >>-
           lfirst2 = .FALSE.
        END IF
!       -<< Executor (L2) >>-
```

Figure 11: Implementation of the REUSE clause

4.9.8.2 Optimizations based on Subscript Analysis

Other optimizations of VFC in the context of independent loops are based on symbolic analysis of access patterns, halos, and the `RESIDENT` clause.

After construction of the compile-time array descriptors for all arrays occurring in the loop nest, the compiler attempts to find the most efficient transformation strategy for each of them. This analysis utilizes information provided in the symbol table concerning the attributes of an array (e.g. distribution, shape, resident, halo, etc.), and information obtained from the symbolic analysis of access patterns. Separately for dimension of an array, each subscript is analyzed in order to determine its transformation scheme. Currently the compiler distinguishes two classes of subscripts:

- *Local subscripts*
  For such subscripts the compiler can proof that no communication is required. An array subscript in a particular dimension is local, either if the array is mentioned in the `RESIDENT` clause, or if the array has the same shape, distribution and, in the corresponding dimension, the same subscript expression as the variable mentioned in the `ON HOME` clause.

- *Non-Local subscripts*
  These are all subscripts which may require communication.

Local subscripts are processed according to one of the following transformation schemes:

1. TR_SIMPLE
   In this case no runtime analysis, no communication, and no transformation of array subscripts is performed. To apply this strategy the following conditions must be satisfied:

   - The corresponding array dimension has a BLOCK or GEN_BLOCK distribution and
   - the subscripts is equal to the loop iteration variable.

   An example for such a reference is A(j,i) in Figure 4.

2. TR_LINEAR
   In this case no runtime analysis and no communication is required. However, a linear transformation of the subscripts is required in order to convert global to local indices. This strategy is applied if

   - The array has a BLOCK, GEN_BLOCK or CYCLIC distribution and
   - the subscript in each dimension is a linear expression of the loop iteration variable.

3. TR_LOCAL, TR_RESIDENT
   In this case no runtime analysis and no communication is required, however the subscripts are transformed into indirect subscripts. An example of such a transformation is array D in Figure 4. The transformed reference is shown in Figures 6 and 7.

Non-local references are processed by applying the general inspector/executor strategy: TR_INSPEX. For example, this is the case for arrays X and B in Figure 4 and the corresponding inspector/executor code in Figure 6, 7.

4.9.8.3 Optimizations based on Halos

Arrays which have the *halo region* specified are processed in special way. Since the global array subscripts accessed in an independent loop by a particular processor must always represent a subset of the halo, the inspector code and the transformation from global to local indices can be accomplished with a much simpler mechanism than for distributed arrays without a halo.

Moreover, since with the current implementation of halos it is always the responsibility of the user to update halos by means of the UPDATE_HALO directive, in the executor no calls to communication routines from the runtime library are generated.

An example of such a transformation can be seen for array C of Figure 4 in Figures 6 and 7.

4.10 Transformation of Scalar Assignments

Scalar assignments to elements of a distributed array which do not occur within independent loops are transformed in such a way that they are executed on the processor that owns the corresponding array elements. If in addition on the right-hand-side of such assignments array elements that might be located on a different processor are referenced, communication statements have to be generated in front of the assignment statements in order to communicate these array elements from their owners into temporary variables on the executing processors. The assignment statements are transformed by converting the access to the left-hand-side array element such that the global index is transformed into the correct local index using the library function `vhpf_g2l`.

The transformation of scalar assignments outside of independent loops is shown in Example 4.9.

Example 4.9: Transformation of Scalar Assignments

```
!HPF$ DYNAMIC :: A, B, C
      ...
      A(I) = B(J)               ! stmt 1

      A(I) = B(C(I))            ! stmt 2
```

generated code ...

```
      b_tmp = pars_isre(a_dsc, (/i/), b_dsc, b, (/j/))        ! stmt 1
      IF (vhpf_is_local(a_dsc, (/i/))) THEN
        a(vhpf_g2l(a_dsc, i, 1)) = b_tmp
      ENDIF

      c_tmp = pars_ibce(c_dsc, c, (/i/))                      ! stmt 2
      b_tmp_000 = pars_isre(a_dsc, (/i/), b_dsc, b, (/c_tmp/))
      IF (vhpf_is_local(a_dsc, (/i/))) THEN
        a(vhpf_g2l(a_dsc, i, 1)) = b_tmp_000
      ENDIF
```

The first assignment statement is transformed by substituting the right-hand-side array element by a temporary variable `b_tmp`*. The processor that owns array element* `A(I)` *will receive into this temporary variable element* `B(J)` *from the processor that owns it as a result of executing the runtime system routine* `pars_isre`*. The* `IF` *statement ensures that the assignment to* `A(I)` *is only executed by the owner of* `A(I)`*. The function* `vhpf_g2l` *is used to perform the translation of global to local indices depending on the actual distribution of* `A`*.*

For the second assignment statement an additional broadcast communication routine (`pars_ibce`*) is generated to make* `C(I)` *available on each processor.*

This compilation scheme which is also referred to as *single-element strategy* is currently also applied to sequential do loops. If, however, a loop is transformed in this way a huge runtime overhead may be introduced since a single-element message may be generated for each iteration of the loop. In future versions of the system much more efficient parallelization strategies will be provided for the parallelization of regular do loops.

4.11 Transformation of I/O Statements

VFC supports the whole range of I/O features of Fortran 90. I/O statements referencing distributed objects are transformed in such a way that they are executed by a specific processor, called the *io-master*. In case of input statements, the io-master broadcasts the data read to the all other processors according to the distribution of the array. In case of output statements, first all processors that own a portion of the distributed array send their parts to the io-master which then outputs them.

Currently VFC imposes some restrictions on the I/O of distributed arrays: either whole distributed arrays or single elements thereof may be read or written. The I/O of sections of distributed arrays is not yet supported. Both formatted and unformatted I/O is supported but currently only the default format "*" can be used.

I/O statements that do not reference distributed objects are not transformed and thus all features provided by Fortran 90 may be used. I/O statements that reference both distributed and non-distributed objects are split such that each item of the *io-list* is processed separately, and thus no formats other than '*' can be used.

Example 4.10: Transformation of I/O Statements

```
      INTEGER A(M,N), B(M,N), C(N), X(10), S
!HPF$ DISTRIBUTE(BLOCK,*)  :: A, B, C

 s1   OPEN(10, FILE='data.in', FORM='UNFORMATTED', STATUS='OLD')
 s2   READ(10) A
 s3   READ(*,*) S
 s4   READ(*,*) C
 s5   WRITE(*,*) B,S,X                        ! this stmt will be split
 s6   WRITE(*,FMT='(I4)') S+2
```

transformed code ...

```
 s1   IF (vhpf_io_master()) THEN
         OPEN(UNIT=10, FILE='data.in', STATUS='OLD', FORM='UNFORMATTED')
      END IF
 s2   CALL pars_iread_array(a, 10, a_dsc, 0)  ! read (unformatted) distributed array
 s3   IF (vhpf_io_master()) THEN
        READ (5, *) s
      ENDIF
      CALL vhpf_bc_scalar(s)                  ! broadcast scalar
 s4   CALL pars_iread_array(c, 5, c_dsc)      ! read (formatted) distributed array
 s5   CALL pars_iwrite_array(b, 6, b_dsc)     ! write distributed array
      CALL vhpf_write_scalar(s, 6)            ! write scalar variable
      CALL pars_iwrite_rarray(x, size(x), 6)  ! write replicated array
 s6   IF (vhpf_io_master()) THEN              ! write using a format
        WRITE (6, '(I4)') s+2
      ENDIF
```

All I/O statements are executed by the io-master processor. I/O statements involving only non-distributed objects are guarded by an `IF` *statement. The I/O of distributed objects is implemented by calls to VFC's runtime library. I/O statements that reference both distributed and non-distributed objects are split as, for example, the first* `WRITE` *statement (s5) above.*

4.12 Transformation of Modules

A module usually consits of data objects and module-subprograms that operate on these data. If in the *specification-part* of a module `M` processor arrays or distributed arrays are declared, the following transformations are performed.

1. For each distributed array the declaration of a runtime layout descriptor is generated.
2. Processors statements and declarations of distributed arrays are transformed as described in Section 4.2 and 4.3.
3. A logical variable with the name `vhpf_idm_`*nr* is generated and initialized with `false`.
4. A new module-subprogram with the name `vhpf_inm_`*nr* is generated[10], where *nr* is an integer number that is chosen by the compiler. This newly generated module subprogram contains the compiler generated executable statements to initialize processor array descriptors and layout descriptors of distributed arrays declared in module `M` and, in case of statically distributed and statically allocated arrays, an `ALLOCATE` statement to allocate on each processor the local portion of these arrays as described in Sections 4.2 and 4.3 All these statements are guarded by an if-statement such that they are only executed if `vhpf_idm_`*nr* is *true*.
5. In each program unit, in which a use statement for `M` occurs, a call to the generated module subprogram `vhpf_inm_`*nr* is inserted at the beginning of the executable section.

Examples 4.11 and 4.12 show how modules that contain distributed data arrays and how program units that use these modules are transformed.

Example 4.11: Transformation of Modules - Original Code

```
MODULE M

      REAL A(1000)
!HPF$ PROCESSORS R(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE(BLOCK) :: A

END MODULE M


PROGRAM P

   USE M
   ...

END PROGRAM P
```

This example shows a module that declares a processor array and a distributed array which are used in program `P`*. The transformations performed by VFC are shown in Example 4.12.*

[10] If HPF+ programs are compiled with VFC using the separate compilation option `-s`, then the name of the newly generated module-subprogram will be `vhpf_inm_`*name*, where *name* is the name of the module to be initialized. In this case the name of a module must not be longer than 22 characters since Fortran limits the length of names with 31 characters.

Example 4.12: Transformation of Modules - Generated Code

```
MODULE m

      USE vhpf_runtime

!HPF$ PROCESSORS r(1:number_of_processors())

      REAL, ALLOCATABLE :: a(:)
!HPF$ DISTRIBUTE (BLOCK) ONTO r :: a

      TYPE (proc_dsc), POINTER :: r_dsc
      TYPE (rt_dsc_list) :: dlist
      TYPE (rt_dsc), POINTER :: a_dsc

      LOGICAL :: vhpf_idm_0 = .TRUE.

CONTAINS

   SUBROUTINE vhpf_inm_0()            ! generated module initialization procedure

     IF (vhpf_idm_0) THEN
        CALL vhpf_init_proc_dsc(proc=r_dsc, shape=(/number_of_processors()/))
        CALL vhpf_init_rt_dsc(dlist, a_dsc, 2, shape=(/1000/), dtype=(/block/), proc=r_dsc)
        ALLOCATE(a(1:vhpf_extent(a_dsc, 1)))
     ENDIF
     vhpf_idm_0 = .FALSE.

   END SUBROUTINE vhpf_inm_0

END MODULE m


PROGRAM p

   USE m

   ...
   CALL vhpf_inm_0()  ! call initialization procedure for module
   ...

END PROGRAM p
```

This example shows the code generated by VFC for the module and main program of Example 4.11. A new module subprogram `vhpf_inm_0` *containing executable statements to initialize the layout descriptor for the distributed array* `A` *and to allocate on each processor the local portion of* `A`*. This routine is called in every program unit that uses module* `M`*. The initialization statements are executed only* `.TRUE.` *only when* `vhpf_inm_0` *is called for the first controlled by the logical variable* `vhpf_ind_0`*.*

4.13 Extrinsic Interfaces

VFC supports the use of the EXTRINSIC(HPF_LOCAL) and EXTRINSIC(HPF_SERIAL) interfaces. Calls to HPF_LOCAL procedures are transformed such that for a distributed argument on each processor only the local section is transferred. If a distributed argument is passed to a HPF_SERIAL function then the argument is replicated before the call and its original distribution is restored after the call. Moreover, a broadcast operation may be performed by the system for scalar arguments that may have been changed. The transformation of calls to extrinsic procedures is shown in Example 4.13.

Example 4.13: Transformation of Calls to Extrinsic Functions

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

      INTERFACE

        EXTRINSIC(HPF_LOCAL) SUBROUTINE LOCSUB(X)
          INTEGER, DIMENSION(:) :: X
        END SUBROUTINE LOCSUB

        EXTRINSIC(HPF_SERIAL) SUBROUTINE SERSUB(X, S)
          INTEGER :: X(:), S
        END SUBROUTINE SERSUB

      END INTERFACE
      ...

      CALL LOCSUB(A)

      CALL SERSUB(A, S)
```

generated code ...

```
      ...
      CALL locsub(a(1:vhpf_loc_ub(a_dsc, 1):1))  ! call hpf-local function

      CALL vhpf_undist(a, dlist, a_dsc)          ! replicate A
      IF (vhpf_io_master()) THEN
         CALL sersub(a, s)                       ! call hpf-serial subroutine
      ENDIF
      CALL vhpf_restore(a, dlist, a_dsc)         ! restore original distribution
      CALL vhpf_bc_scalar(s)                     ! broadcast scalar
```

The call to the EXTRINSIC(HPF_LOCAL) *procedure* LOCSUB *is transformed such that on each processor only the local section of the distributed array* A *is passed using the runtime system routine* vhpf_loc_ub*. The call to the* EXTRINSIC(HPF_SERIAL) *procedure* SERSUB *is transformed by inserting a call to* vhpf_undist *which replicates* A *on each processor before the invocation of* SERSUB*, an* IF *statement to enforce that it is called on one processor only, and a call to* vhpf_restore *which restores the original distribution after the invocation. Moreover a call to a broadcast routine for the scalar argument* S *is generated.*

5 VFC Runtime Support System

The runtime system of VFC consists of two major components: the VHPF library and the PARS library.

The main tasks of the VHPF library is the management of distributed objects at runtime including initialization of runtime layout descriptors, evaluation of distribution and alignment information, allocation of distributed arrays, provision of original global properties of distributed arrays, and support for dynamic data redistribution. Moreover, the VHPF library provides high level interfaces for the PARS library calls which significantly alleviate code generation.

The PARS library provides low level communication routines utilizing calls to the MPI-1 library, implements the basic I/O procedures, and provides support for all aspects of the inspector/executor runtime parallelization strategy. Moreover PARS provides routines to support the implementation of the new halo concept, including routines to compute from a halo specification the required communication schedules and to update halo areas.

In the following a brief overview of both libraries is given.

5.1 VHPF Runtime Library

The VHPF runtime library has been written in Fortran 90. Its main task is the management of distributed objects at runtime. The library provides a flexible interface to the VFC compiler and supports all aspects of runtime distribution processing, management of distributed objects and their associated runtime layout descriptors, functions for index conversion, ownership computation and dynamic data redistribution. In the following a brief description of the most important VHPF routines is given.

5.1.1 Initialization of Processor Array Descriptors

vhpf_init_proc_dsc() : Subroutine

- **Input**
  - Shape of processor array.
- **Output**
  - Processor array descriptor.
- **Description**
  This function allocates a processor array descriptor, initializes it with the shape information, and generates an MPI communicator for it. Furthermore it performs various checks, for example, if the size of a processor array exceeds the number of available processors.

5.1.2 Initialization of Runtime Layout Descriptors

vhpf_init_rt_dsc() : Subroutine

- **Input**
  - List of existing runtime layout descriptors in program unit.

- Pointer to runtime layout descriptor.
- Element type of corresponding array.
- Shape of array (optional).
- Distribution type (optional).
- Processor array descriptor (optional).
- Processor array section (optional).

• **Output**

- Processor array descriptor.

• **Description**
If both optional arguments *shape* and *distribution type* are present, the list of runtime layout descriptors is searched in order to find a matching runtime layout descriptor, and if one is found a pointer to it is returned. If no matching runtime layout descriptor is found, a new descriptor is generated and initialized. If one of these optional arguments is not present, the layout descriptor is partially initialized or updated.

5.1.3 Data Redistribution

vhpf_redist() : Subroutine

• **Input**

- Array to be redistributed.
- List of existing runtime layout descriptors in program unit.
- Actual runtime layout descriptor.
- Temporary runtime layout descriptor.

• **Output**

- Redistributed array.
- Updated runtime layout descriptor.

• **Description**
This subroutine redistributes an array according to the new distribution of the temporary runtime layout descriptor. Usually this is done be allocating memory for the array according to the new distribution, performing the transfer of physical data from the array to the newly allocated memory and updating the array's runtime layout descriptor with the new distribution. If the array to be distributed has not yet been allocated, no physical data transfer takes place.

vhpf_undist() : Subroutine

• **Input**

- Array to be redistributed.

  - List of existing runtime layout descriptors in program unit.
  - Actual runtime layout descriptor.

- **Output**
  - Replicated array.

- **Description**
  This subroutine replicates a distributed array such that it is allocated on each processor.

vhpf_restore() : Subroutine

- **Input**
  - Replicated array.
  - List of existing runtime layout descriptors in program unit.
  - Runtime layout descriptor of array to be restored.

- **Output**
  - Distributed Array.

- **Description**
  This subroutine distributes a replicated array according to the distribution of the layout descriptor.

5.1.4 Work Distribution

vhpf_work_dist() : Subroutine

- **Input**
  - Runtime layout descriptor.
  - Dimension number.
  - Description of accessed array section.

- **Output**
  - Description of work set (section or index array) on executing processor.

- **Description**
  This subroutine computes the intersection of the accessed array section with the local index set on the executing processor.

5.1.5 Assignment of Array Sections

vhpf_sec_assign() : Subroutine

- **Input**
  - Runtime layout descriptors of lhs and rhs array.
  - Descriptions of accessed lhs and rhs array sections.
  - Handle to communication schedule (optional).
- **Output**
  - Handle to communication schedule (if present).
- **Description**
  This subroutine performs the assignment of an arbitrary lhs array section to an arbitrary rhs array section. Currently this routine calls the function `remap` from the ADLIB library citeadlib.

5.2 PARS Runtime Library

The PARS (PArallel Runtime System) library is a successor of the PARTI [21] library to support the parallelization of irregular programs, i.e. programs that are characterized by data distributions and access patterns that are dependent on runtime data and thus cannot be fully analyzed at compile time. The PARS library constitutes a completely new development which introduces the following novel features :

- Provision of standard as well as non-standard data distributions, including general-block and indirect distributions.
- Generating schedules for multi-dimensional arrays which are arbitrarily distributed in more than one dimension.
- Dimensional processing of accesses to multi-dimensional arrays which simplifies the inspector/executor code generation.
- Supporting single-element communication and simple I/O for distributed arrays.

Note that most of the routines described in the following are overloaded for different F90 data types by the VHPF library.

5.2.1 Schedule generation

A schedule is a data structure which describes the communication pattern of independent loops. Schedules are computed in the inspector phase of independent loops and are used to gather and scatter non-local data between the processors. In the following we shortly introduce the different routines of PARS which are responsible for schedule generation.

dview() : Subroutine

- **Input**
  - Runtime descriptor of the mapped data array for which schedule will be generated.
  - Global index (stored in GBOX derived type parameter) references to the data array on each processor.
- **Output**
  - Equivalent local references of the global indices on each processor (returned in LBOX derived type parameter).
  - Number of non-local references in each dimension.
  - Schedule for the communication pattern of the distributed array in the irregular loop.
- **Description**
  After localizing each global reference which in fact identifies the owner processor and the local offset on this processor - by means of a hashing mechanism - **dview()** builds lists of owning processors and the local offsets on the corresponding owner processors for all non-local references while filtering out the repeated non-local references. These two lists are passed to **schedulen()** which continues further the process of schedule generation.
  The returned array containing localized references (the first one in the **Output** parameter list) contains for every global reference which denotes a local element of the data array, the corresponding index from local index space of the data array and for every global reference which denotes a non-local element, the index of the next free element of the allocated buffer.

Schedulen() : function

- **Input**
  - Mapping descriptor of the distributed data array and the processor array onto which the data array is distributed.
  - Array containing the indices of the non-local elements from the local index space on the owner processor.
  - Array containing the processor number of the owner of each non-local reference.
  - The size of the two previous arrays.
- **Return Value**
  - Structure describing the communication schedule.
- **Description**
  It calls **bsort()** to sort the non-local elements into different buckets. Each bucket corresponds to a processor and contains those references which are local on this processor. Then it passes the buckets to **schedule_gather()** which initiates a global communication in that every processor informs other processors which elements it needs from them while it receives from others which elements they need. In fact this information describes the content of the schedule.

bsort() : Subroutine

- **Input**
  - Number of allocated processors.
  - Number of non-local references.
  - List of owner processors of the non-local elements.
  - List of the offset of the non-local elements on the owner processor.
- **Output**
  - Array of size *nproc*. Each element i contains the number of non-local elements that are local on processor P_i .
  - Array of size *nproc* (array of buckets). Each element i contains those of non-local elements that are local on processor P_i .
  - Array of size *nproc*. Each element i contains the indices of elements of the buffer (attached to data array) which will contain those non-local elements that are local on processor P_i .
  - Array of size *ndata* (number of non-local elements). This one dimensional array is simply a linear view of the buckets array (second **Output** parameter).
  - Array of size *ndata* (number of non-local elements). This one dimensional array is simply a linear view of the third **Output** parameter.
- **Description**
  This routine divides the non-local references among different buckets (each representing a processor) by means of bucket sort. The order of appearance of the non-local references in the **Input** array which in fact denotes the index of the buffer (in that the non-local elements will be gathered or from it scattered to other processors) is also put in to a separate array. This array is later used in **gather()** and **scatter()** to put the communicated non-local elements into the correct buffer cells. This array is also part of schedule.

schedule_gather() : function

- **Input**
  - Mapping descriptor of the distributed data array and the processor array onto which the data array is distributed.
  - Array of size *nproc*. Each cell i of this array contains the list of non-local elements which are local on processor P_i .
  - Array of size *nproc*. Each cell of this array contains the size of the list in corresponding cell of the previous **Input** array.
- **Return Value**
  - Structure defining the communication pattern to be used in the *execution* phase.

- **Description**
It starts a global communication among the allocated processors. Each processor sends to the others the indices of non-local data that it needs for the *execution* phase and receives from each processor the list of element-indices that it needs.
These two exchanged lists of indices (i.e. the *send* and *receive* lists) define indeed the communication pattern.

5.2.2 Communicating non-local elements

The PARS routines, **gather()** and **scatter()** are the two routines which upon invocation perform the communication described in the schedule passed to them. As the functionality of these two is very similar, we will introduce only one of them.

gather() : subroutine

- **Input**
  - Runtime descriptor of the data array.
  - Address of the data array.
  - Schedule (generated in a previous phase).
  - Size of data unit to be exchanged (the communication is done in bytes).
- **Output**
  - The updated buffer, that is appended to the local section of the array on each processor.
- **Description**
This routine first invokes **execute_gather()** which performs the communication described in the **Input** schedule and then by means of **bytecopy()** routine copies the communicated elements into the target buffer.

execute_gather() : subroutine

- **Input**
  - Schedule of the distributed array.
  - A pointer to the local section of the distributed array.
- **Output**
  - A pointer to the temporary buffer, allocated in **gather()**.
- **Description**
It copies the elements which should be sent to different processors, into a temporary buffer and sets the necessary parameters of the **MPI_Alltoallv()** and calls it to do the communication.

5.2.3 Halos

Halo is a region in buffer of a distributed array that contains all non-local references of the array on a specific processor for the life-time of the array. Therefore schedule is generated once for an array with Halo for its life-time.

s4h() : Subroutine

- **Input/Output**
  - Runtime descriptor of the mapped data array for which schedule will be generated.
- **Description**
  s4h() receives the Halo elements for each dimension in a derived type variable named MAX_GBOX stored in the runtime descriptor. For each dimension it also adds local index set (in their global form) on each processor to MAX_GBOX and sorts all references to each dimension. This routine also makes index conversion for the whole references in MAX_GBOX and stores the result in MAX_LBOX which is also a field in runtime descriptor. Finally s4h() generates a schedule for the non-local references and store it in the runtime descriptor.

update_halo() : Subroutine

- **Input**
  - Runtime descriptor of the mapped data array for which schedule will be generated.
  - Address of the data array.
  - Schedule (generated in a previous phase).
- **Output**
  - The updated buffer, that is appended to the local section of the array on each processor.
- **Description**
  This routine performs the communication described in the **Input** schedule and then by means of **bytecopy()** routine copies the communicated elements into the target buffer.

halo_indxmapr() : Subroutine

- **Input**
  - Runtime descriptor of the mapped data array for which schedule will be generated.
  - Global index references to the data array on each processor.
- **Output**
  - Equivalent local references of the global indices on each processor.

- **Description**
  For each dimension of the data array and each global reference in the GBOX (input parameter) this routine searches the MAX_GBOX (stored in runtime descriptor) to find the global index then uses its reference index in the MAX_GBOX to get the equivalent local index MAX_LBOX and sets the appropriate element of LBOX output parameter with this retrieved local index.

5.2.4 Element-wise Communication

The two routines SRE (Send Receive Element) and BCE (BroadCast Element) support the implementation of element-wise communication strategy for parallelizing complex loops that other parallelization strategies fail to handle.

sre() : function

- **Input**
  - Layout descriptor of the target array.
  - Global references to an element of the target array.
  - Layout descriptor of the source array.
  - Local section of the source array.
  - Global references to an element of the source array.
- **Return Value**
  - Value of the referenced element of the source array or an undefined value (depending on the calling processor).
- **Description**
  This function simply returns the value of the referenced element of the source array to the processor that owns the referenced element of the target array. The returned value to every other processor is undefined.

bce() : function

- **Input**
  - Layout descriptor of the distributed array.
  - Local section of the distributed array.
  - Global references to an element of the distributed array.
- **Return Value**
  - Value of the referenced element of the source array.
- **Description**
  This function returns the value of the referenced element of the distributed array to the calling processor.

5.2.5 I/O of Distributed and Replicated Arrays

Although I/O is not part of PARS, we introduced simple support for distributed and replicated arrays for the sake of completeness. Here we only describe input routines. The output routines have similar interfaces.

read_rarray() : subroutine

- **Input**
  - The replicated array.
  - Number of elements of the array.
  - Number of the input unit.
  - A flag indicating whether to perform a formatted or unformatted read.
- **Output**
  - Updating the array with the input.
- **Description**
  After calling this routine, the I/O master (usually processor number zero) reads from the specified unit and broadcasts to all the other processors.

read_array() : subroutine

- **Input**
  - The local section of the distributed array.
  - Number of the input unit.
  - Layout descriptor of the distributed array.
  - A flag indicating whether to perform a formatted or unformatted read.
- **Output**
  - Updating the array with the input.
- **Description**
  After calling this routine, the *io-master* (usually processor number zero) reads from the specified unit and broadcasts to all the other processors. Then each processor copies from the communicated message that part that it owns into its local section.

5.2.6 Optimizations

It happens very frequently that an independent loop contains several occurrences of a specific distributed array. In this case the collected global references for this array would contain repeated non-local references. The naive buffer allocation approach would be assigning to each non-local reference in GBOX a separate buffer cell. But this straight forward strategy has two serious disadvantages. First the allocated buffer for the distributed array may become larger than the global size of the array. Second this will result also to redundant communication for the repeated non-locals. Dview() now hashes out the repeated non-local references and assigns unique buffer spaces for repeated non-locals. The generated schedule also contains each non-local data element only once and hence any potential redundancy in the communication is also eliminated.

6 Instrumentation

In this section we describe the instrumentation system (SIS), which allows the automatic and selective measurement of codes parallelized by VFC, by automatically inserting, based on command-line options, calls to a measurement runtime library. As the result of the execution of an instrumented HPF+ program a tracefile with the measurement information collected at runtime is obtained which then can be analyzed by post-execution performance analysis tools such as, for example, MEDEA.

The instrumentation system provides the application developers as well as the compiler developers with source–level information about the performance of parallelized codes. The main objective of the instrumentation system is to detected the sources of performance losses which may be due to the algorithm or data structures employed by the HPF+ program or due to the parallelization transformations performed by the compiler can be identified.

The instrumentation system has been designed to support the measurement of important code regions such as subroutine calls, INDEPENDENT loops, sequential loops, single statement and I/O operations. The code regions to be instrumented may be selected by the user by means of command-line options.

6.1 Overview

The instrumentation system consists of three main componenets,

- the *code region selector*,
- the *sequential code instrumenter*, and
- the *parallel code instrumenter*,

which are tightly integrated within the VFC compiler.

The *code region selector* is dedicated to analyzing the HPF+/HPF codes submitted to the VFC, and following user directives given in the command line, identifies the code regions to be instrumented. This process is needed to collect compiler-internal information regarding the instrumented code region.

The *sequential code instrumenter* may be used to instruments sequential Fortran codes, which can be executed on a sequential machine.

The *parallel pode instrumenter* represents the most important component of the instrumentation system and is fully integrated within the various parallelization techniques applied by the VFC.

6.2 Automatic Instrumentation

The instrumentation system inserts in the parallel code calls to runtime measurement library for generating tracefiles in several different formats, allowing the use of various different performance analysis tools.

Different tracefiles can be generated without the need of recompiling the code. The current implementation of the instrumentation system allows to generate tracefiles according to the ALOG format and an internal tracefile format. The selection of the tracefile format can be controlled by the user by setting corresponding environment variables.

For each code region to be instrumentated a corresponding measurement statement is inserted at the beginning and at the end of the code region, repsectively. An example of the instrumentation of a subroutine call is given in Figure 6.1.

Example 6.1: Instrumentation of a Subroutine Call

```
CALL vhpf_measure(1096, vhpf_se, vhpf_beg_meas, 0, -1, -1)
CALL sub1(xn,yn)
CALL vhpf_measure(1097, vhpf_se, vhpf_end_meas, 0, -1, -1)
```

The first parameter of the measurement rouitne `vhpf_measure` *is the measurement point - an integer identifying univocally the measurement region; the second parameter is a label describing the code type of region measured. In the above example* `vhpf_se` *represents a subroutine call, the third parameter can be the beginning* `vhpf_beg_meas` *or the end* `vhpf_end_meas` *of the region.*

The instrumentation system inserts a call to the measurement library routine `vhpf_sis_init()` at the beginning of a program to initialize the instrumentation. Similarly, at the end of a program a call to the routine `vhpf_sis_final()` is inserted at the end of a program in order to finalize the instrumentation environment and to write the tracefiles.

Instrumentation of `INDEPENDENT DO` Loops

VFC transforms independent `DO` loops based on the inspector-executor paradigm by combining compile time parallelization techniques with runtime parallelization techniques. Since for an independent loops the user asserts that there are no loop-carried dependences, that is, the execution of an iteration does not change data items used in other iterations, any communication can be performed before and after the loop. The parallelization of a independent loop can be described as a sequence of separate phases: *work distribution, inspector, gather communication, executor* and *scatter communication.* During the process of transforming an independent loop according to these phases, the instrumentation system instruments each phase allowing a precise performance analysis of independent loops.

6.3 Manual Instrumentation

Any statement or arbitrary sequences of executable statements can be instrumented by manually inserting in source code the directives `SIS_MEASURE_BEGIN` and `SIS_MEASURE_END` for identifying the beginning and the end of a code region to be mesured. As shown in Example 6.2 these directives must start with the prefix `!VFC$`. During the parallelization, SIS substitutes the user directives with measurement statements. This allows to instrument any piece of code containing executable statements and any further modification of the source code does not effect the instrumentation. The portability of the code is maintained since the measurement directives are treated as comments by any other compiler.

Example 6.2: Instrumentation Directives

```
!VFC$ SIS_BEGIN_MEASURE

      ...

!VFC$ SIS_END_MEASURE
```

6.4 Relating Performance Data to the HPF Level

One of the most important features required in todays performance analysis tools is to relate performance indices to the user source code. The instrumentation system address this issue by keeping trace of the parallelization steps performed by the VFC compiler and providing the following measurement description for each code region measured.

- Type of region measured (e.g. subroutine, loop, independent loop, etc.)
- Name of the input HPF+ file containing the code region.
- Original source code postion. (i.e. line and column number of begin and end positions in the original source code)
- The subroutine or function name in case that a procedure has been instrumented.
- The measurement points inserted.

A *measurement description file* is shown in Example 6.3.

Example 6.3: Measurement Description

```
P  f1hpf+.hpf     1  7   112 10 HJHPF.1.PR 1000  1001
S  f2hpf+.hpf    96 12    96 22 TCHECK.96.SE 1054  1055
X  kfrq.hpf    30  7   51  7 FORCE.30.IE 1068  1069
T  kfrq.hpf    30  7   30 69 FORCE.30.IS 1066  1067
G  kfrq.hpf    30  7   30 69 FORCE.30.IG 1064  1065
N  kfrq.hpf    30  7   30 69 FORCE.30.II 1062  1063
D  kfrq.hpf    30  7   30 69 FORCE.30.IW 1060  1061
S  kfrq.hpf    41 14   41 30 JFORCE.41.SE 1080  1081
S  wtime.hpf     5 12    5 33 SYSTEM_CLOCK.5.SE 1086  1087
```

A one-to-one correspondence exists between the positions - line and column number in the source code - and the measurements points of the parallel code. As an example, the call to the subroutine TCHECK of Figure 6.3 is at line 96 in file f2hpf+.hpf and between measurement point 1054 and 1055 in the generated parallel code.

The main purpose of the measurement description is to establish a one-to-one correspondence between the source code and the measurements points of the parallel code. By combining the measurement description obtained at compile time with the tracefile produced during runtime, MEDEA is capable of providing detailed performance information at the HPF+ source code level.

6.5 Other Features

The instrumentation system has also been designed in order to investigate the status of any distributed array at any point in the code. This feature is particularly useful, for example, to obtain detailed information on the state of dynamically distributed arrays at various points in the program. The fact that the compiler generates a layout descriptor for each distributed array significantly alleviates this process. By inserting during the program transformation calls to the runtime control procedure `vhpf_build_control()`, information on the actual distribution of an array at a particular point during runtime may be obtained. According to the information contained in the layout descriptor of a distributed array this includes information about the the distribution of the array, the global shape information, the local shape of the array on a particular processor, the sizes of buffers alloacted to store copies of non-local data, and information about communication schedules, for example, to update the halo of a distributed array (if present).

7 Using VFC

In this Section we briefly describe how VFC can be used to parallelize and executed HPF+ programs on the QSW CS-2 and the NEC Cenju-3. For more information on using VFC consult the HPF+ User's Guide.

7.1 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.

`-m[cils]` Insert instrumentation calls.
`c` ... instrument communication statements (default)
`i` ... instrument `INDEPENDENT DO` loops
`l` ... instrument sequential `DO` loops
`s` ... instrument `SUBROUTINE` calls
At least one of the above options 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. (May be used when a program stops with the message: "DVIEW: not enough space in buffer - allocation ration *n*")

`-h` Enable the use of halos.

7.2 QSW CS-2

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

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

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

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.

7.3 NEC 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
```

8 Conclusion

In this document we described the HPF+ compilation system VFC which has been developed within the HPF+ project. Compared to its predecessor VFCS, VFC is based on full Fortran 90 and supports a number of important features like dynamic memory allocation, dynamic data distribution, parameterization of programs by the number of processors, communication schedule reuse, halos, separate compilation and the Fortran 90 modularization concepts such as internal procedures, interfaces, and modules.

The main focus with respect to the development of parallelization techniques has been on independent loops with irregular runtime dependent data distributions and access patterns. In this context the inspector/executor runtime parallelization technique has been significantly generalized such that is applicable to non-perfectly nested loops that access arbitrarily distributed multi-dimensional arrays. To amortized the overheads introduced by runtime preprocessing of such loops the concept of schedule reuse has been implemented. This feature has shown to be crucial for achieving acceptable performance of irregular codes. Moreover, the concept of halos, allows the user to provide the compiler with high level information about non-local irregular data accesses and the associated communication schedules, resulting in an improved efficiency of the generated codes, both in terms of memory requirement and execution time.

Besides advanced parallelization techniques for nested independent loops, VFC also provides support for an efficient parallelization of array assignments. However, VFC does currently not provide optimized parallelization techniques for `DO` loops that exhibit only regular data access patterns. Such constructs are currently parallelized by applying the before mentioned runtime parallelization techniques for irregular loops and as a consequence a large overhead may be introduced. Moreover, in order to handle dynamic concepts of HPF like dynamic data redistribution, it was necessary to develop a very general parallelization strategy. This may result in additional overheads for codes that are inherently static. Optimization techniques for regular codes and inherently static distributions have not been the main focus of this project but are planned to be implemented in future versions of VFC.

The following contributions to the development of HPF compilation techniques have been made within the HPF+ project.

- An almost complete implementation of HPF+ including *dynamic concepts* such as dynamic data redistribution has been realized. VFC not only supports fine grain parallelism in data parallel constructs and independent loops, but also coarse grain parallelism through the provision of the `HPF_LOCAL` extrinsic mechanism.
- New data distribution formats required for irregular codes, such as *generalized block* and *indirect distributions*, currently not available in any HPF compilation, have been implemented.
- The inspector/executor approach has been generalized to nested independent loops with highly irregular accesses to multi-dimensional distributed arrays.
- A number of optimization techniques for the inspector-executor parallelization technique have been developed and implemented.
- The important concept of *communication schedule reuse* based on language support has been developed and implemented.

- The *halo concept*, a facility for specifying at a high level irregular non-local data access patterns has been developed and implemented. Halos not only alleviate the allocation of temporary storage for non-local data but may also significantly reduce the overheads of communication schedule computation and irregular communication.
- Other language features to avoid potential communication overheads such as the `RESIDENT` clause for independent loops and `PUREST` procedures have been implemented.
- A powerful runtime support system based on MPI has been designed and implemented.

9 Acknowledgements

The authors would like to thank the other partners of the HPF+ project for their valuable feedback on the VFC compiler and for the many fruitful discussions and suggestions to improve the system.

References

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

[2] S. Barros, D. Dent, L. Isaksen, G. Robinson, G. Mozdzynski, and F. Wollenweber. The IFS model: A parallel production weather code. Parallel Computing 21, 1995.

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

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

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

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

[7] S. Benkner. A User's Guide for HPF+. Deliverable D2.1d of the ESPRIT LTR Project HPF+. University of Vienna, March 1998.

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

[9] S. Benkner, H. Zima. Definition of HPF+ Release 2. Deliverable D2.1b of the ESPRIT LTR Project HPF+. University of Vienna, February 1997.

[10] S. Benkner, H. Zima. HPF+ Initial Language Definition. Deliverable D2.1a of the ESPRIT LTR Project HPF+. University of Vienna, June 1996.

[11] S. Benkner, et.al. Vienna Fortran Compilation System. Version 1.2. User's Guide, Institute for Software Technology and Parallel Systems, February 1996.

[12] S.Benkner. Vienna Fortran 90 - An Advanced Data Parallel Language. In *Proceedings of the third International Conference on Parallel Computing Technologies*, PaCT-95, St.-Petersburg, Russia, September 1995, Springer Verlag.

[13] S. Benkner. Handling Block-Cyclic Distributed Arrays in Vienna Fortran 90. In Proceedings of the 'International Conference on Parallel Architectures and Compilation Techniques', PACT'95, Limassol, Cyprus, June 1995.

[14] 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.

[15] S. Benkner, P. Brezany, H. Zima. Processing Array Statements and Procedure Interfaces in the Prepare High Performance Fortran Compiler. In P.A. Fritszon, editor, *Compiler Construction, Proceedings of the 5th International Conference*, LNCS 786, pages 324-338. Springer-Verlag, April 1994.

[16] S. Benkner, P. Brezany, H.P. Zima. Compiling High Performance Fortran in the Prepare Environment. In *Proceedings of the Fourth International Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, 1993

[17] S. Benkner, B. Chapman, and H. Zima. Vienna Fortran 90. In *Proceedings of the SHPCC Conference 1992*, Williamsburg, USA, pp. 51-59, April 1992.

[18] B. Chapman, P. Mehrotra, and H. Zima. Vienna Fortran - A Fortran language extension for distributed memory systems. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-time Environments for Distributed Memory Machines*. Elsevier Press, 1992.

[19] J. Clinckemaillie, B. Elsner, G. Lonsdale, S. Meliciani, S. Vlachoutsis, F. de Bruyne, M. Holzner. Performance Issues of the Parallel PAM-CRASH Code. Journal of Supercomputing Applications and High-Performance Computing, Vol.11, No.1, 1997.

[20] High Performance Fortran Forum.

[21] R. Das, J. Saltz, A manual for PARTI runtime primitives - Revision 2. Internal Research Report, University of Maryland, Dec. 1992

[22] J. Saltz, R. Das, B. Moon, S. Sharma, Y-S. Hwang, R. Ponnusamy, M. Uysal : *A Manual for the CHAOS Runtime Library*, Technical Report, University of Maryland, College Park, MD 20742, May 1994.

[23] H. M. Gerndt. Automatic Parallelization for Distributed-Memory Multiprocessing Systems. PhD thesis, University of Bonn, December 1989.

[24] H.M. Gerndt. Array Distribution in SUPERB. In *Proceedings of the 3rd International Conference on Supercomputing* 1989, 164-174.

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

[26] H.M. Gerndt. Work Distribution in Parallel Programs for Distributed Memory Multiprocessors. In *Proceedings of the International Conference on Supercomputing*, 1991.

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

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

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

[30] J. Knoop and E. Mehofer. Interprocedural distribution assignment placement: More than just enhancing intraprocedural placing techniques. IEEE Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT'97) San Francisco, CA, November 1997.

[31] D. B. Carpenter. Adlib: PCRC Runtime Kernel. see: http://www.npac.syr.edu/projects/pcrc/adlib.html

[32] C. Koelbel. *Compiling Programs for Nonshared Memory Machines.* PhD thesis, Purdue University, West Lafayette, IN, August 1990.

[33] 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.

[34] J. Saltz, H. Berryman, and J. Wu. Runtime compilation for multiprocessors (To appear: Concurrency, Practice and Experience, 1991). ICASE Report 90-59, ICASE, 1990.

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

[36] H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

[37] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series, Addison-Wesley, 1990.

[38] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, A. Schwald. *Vienna Fortran - A Language Specification, Version 1.1* Technical Report ACPC/TR 92-4, Austrian Center for Parallel Computation, March 1992 Also published as: NASA Contractor Report 189629, ICASE Interim Report 21, NASA Langley Research Center, Hampton, Virginia, March 1992.

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-specs* 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-dummys* 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.