Extension of VFCS for the Compilation of Project Benchmarks Vers. 2

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

Deliverable: D2.2b
Date: September 1997
Status: draft
Confidentiality: public
Responsible: University of Vienna

Revised: September 19, 1997

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

©1997 by the **HPF+** Consortium

Contents

1 Introduction **1**

2 Overview of VFC **2**
2.1 Major Aspects of VFC . . . 2
2.2 Data Parallel Language Features . . . 3
2.3 Fortran 90 Language Features . . . 4
2.4 Overview of the Parallelization Strategy . . . 5
2.5 System Structure . . . 7

3 HPF and HPF+ Language Features **9**
3.1 Processor Arrays . . . 9
3.2 Data Distribution Mechanisms . . . 9
3.3 Runtime-defined Static Distributions . . . 11
3.4 Dynamic Memory Allocation . . . 11
3.5 Dynamic Data Redistribution . . . 12
3.6 Distribution of Dummy Arrays and Local Arrays . . . 13
3.7 The `INDEPENDENT` Directive . . . 14
3.8 Pure and Purest Procedures . . . 16
3.9 Extrinsic Interfaces . . . 18
3.10 Fortran 90 Features . . . 18

4 Compilation **19**
4.1 Processing of `PROCESSORS` Directives . . . 19
4.2 Management of Distributed Arrays . . . 20
4.3 Processing of Function Calls . . . 24
4.4 Processing of Procedure Calls and Procedure Interfaces . . . 25
4.5 Processing of I/O Statements . . . 26
4.6 Processing of Scalar Assignments Outside Independent loops . . . 27
4.7 Parallelization of Array Assignments . . . 28
4.8 Parallelization of Independent Loops . . . 29
4.9 Extrinsic Interfaces . . . 42

5 VFC Runtime Support System **43**
5.1 VHPF Runtime Library . . . 43
5.2 PARS Runtime Library . . . 45

6 Using VFC **51**
6.1 Command Line Syntax . . . 51
6.2 Restrictions . . . 52

7 Conclusion and Future Work **54**

A Syntax **57**
A.1 Syntax of Directives . . . 57

A.2 The PROCESSORS Directive 58
A.3 The DISTRIBUTE Directive 59
A.4 The INHERIT Directive 59
A.5 The INDEPENDENT Directive 60
A.6 Reduction Statements 60
A.7 The DYNAMIC Directive 61
A.8 The REDISTRIBUTE Directive 61
A.9 Function and Subroutine Statements 61

1 Introduction

In this report we describe the new release of the HPF/HPF+ compiler VFC which has been developed for the parallelization of the second version of the HPF+ project benchmarks.

VFC is a command-line parallelization system that translates Fortran90/HPF/HPF+ programs to Fortran90/MPI message passing programs. The system is currently available on the Meiko CS-2 and on clusters of workstations. It can be ported to any parallel system that provides a full Fortran 90 compiler and an MPI implementation.

Whereas the VFCS which has been used in the project so far was based on Fortran 77 with support for some Fortran 90 extensions, VFC accepts full Fortran 90/HPF 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 has been developed.

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 outline the use of the system and future work.

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 that have been implemented for the parallelization of irregular applications. Section 5 gives an overview of the runtime support system. Section 6 describes how VFC may be used to parallelize and execute HPF programs. Section 7 summarizes the major achievements and outlines the future work planned until the end of the HPF+ project.

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 Meiko CS-2 (the project platform) and on clusters of SUN workstations (the development platform).

Input languages for VFC are HPF-2 [21] with HPF+ [10] extensions and many of the language features specified in the HPF-2 Approved Extensions.

The parallelization strategy offered by VFC is much more general and flexible than that provided by VFCS. It is applicable 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.

A 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*.

VFC implements most of the mechanisms required by HPF-2 and its Approved Extensions for the transfer of distributed arguments across procedure boundaries and supports the `HPF_LOCAL` and `HPF_SERIAL` interfaces.

A detailed description of the various parallelization strategies of VFC can be found in Section 4.

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 HPF-2 distribution mechanisms (`BLOCK, CYCLIC`, block-cyclic distributions, replication), 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.

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.

VFC provides the following data distribution features:

- *Regular block and cyclic distributions*
- *General block distributions*
- *Indirect distributions*
- *Replication*
- *Multidimensional distributions*
- *Inheritance of distributions*
- *Distribution of* `ALLOCATABLE` *and* `POINTER` *objects*
- *Dynamic redistribution*

Independent Loops

The INDEPENDENT directive may be used to assert to the compiler that the iterations of a 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, the ON and ON HOME clauses which enable the user to specify how the loop iterations should be distributed, and the REUSE clause for indicating that the communication schedules of a loop can be used (possibly depending on a logical condition). A detailed description of INDEPENDENT loops as currently implemented in VFC is given in Section 3.7.

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. It is planned to implement at least a number of basic functions of this library, for example, MY_PROCESSOR, GLOBAL_SIZE and GLOBAL_SHAPE.

2.3 Fortran 90 Language Features

In the following we summarize important Fortran 90 features offered by VFC which were not supported by VFCS.

Dynamic Memory Allocation

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

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 mapped arguments at procedure boundaries. In particular, VFC does not support reshaping of mapped arrays and the transfer of sub-arrays.

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

The distribution of dummy arrays may be specified either with the `INHERIT` attribute or with the `DISTRIBUTE` attribute, in which case the actual and formal 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. Implied do-list I/O is also not available 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.

[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.

Moreover, it is possible to incrementally update layout descriptors, which significantly simplifies the implementation of dynamic memory allocation and dynamic data redistribution.

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 mapped actual argument is associated with a non-mapped 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, 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 mapped actual argument is associated with a non-mapped 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

Currently VFC efficiently parallelizes only a restricted class of array assignment statements that operate only on arrays with identical distributions.

VFC analyzes all array assignment statement that involve distributed objects to determine if they can be parallelized without the need for introducing inter-processor communication. If the system can proof that no communication will be required it transforms the array assignment statements in such a way that each processor only accesses data local to it.

Opposed to VFCS, array statements are not converted to loops internally, but are transformed to array statements that operate on the local section on each processor only. The main advantage of this strategy is that the parallelism of the original construct is preserved. Therefore, inter- as well as 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 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. 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 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 the actual computation is performed without the need for non-local data.

2.5 System Structure

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 frontend mainly consists of three components, the scanner, the parser and the tree-builder. These components have been implemented based on the Cocktail tools REX, LALR and AST, respectively. As already mentioned, the frontend has been initially developed by GMD Berlin and has been extended within this project to support the HPF+ 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 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 (their description is beyond the scope of this document).

Moreover, a flow graph of the program is constructed which will be the basis for the integration of a powerful dependence analysis test and the planned interprocedural analysis framework which is currently under development at Vienna.

The *parallelization component*, which transforms the input program into a parallel message passing SPDM program consists of several components. The first is the *normalization component* which normalizes the code and performs various transformations such as the substitution of function calls from expression that reference distributed objects, and the normalization of procedure interfaces. The task of the *classification component* is to determine for each executable construct the parallelization strategy that should be applied to it. Currently three different parallelization strategies are available. The inspector/executor strategy is utilized for parallelizing independent loops. A regular parallelization technique is applied to array assignments and has shown to be very efficient for initialization of distributed arrays or for array assignments that operate on distributed arrays that are identically distributed. A default strategy, based on single-element send/receive/broadcast operations, is applied to sequential or non-parallelizable loops. All these parallelization strategies have been implemented based on the PUMA tool.

The *instrumentation component* has the task of inserting calls to a measurement library for generating trace-files during execution of the program. The instrumentation can be triggered by the user by means of command-line options, and supports selective program instrumentation, e.g. for the different phases of independent loops.

The *code generator* translates the internal representation of the program into Fortran 90 source code, which then is submitted to a Fortran 90 compiler (currently SUN F90 1.2) for generating the executable parallel program.

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 intermediate language definition [10].

3.1 Processor Arrays

Whereas VFCS required that the number of processors a program should be parallelized for is fixed at compile-time, 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 realized. 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. The syntax of the `PROCESSORS` statement as currently implemented is specified in the Appendix. 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. Since the* `RESHAPE` *intrinsic function does not contain a shape expression (see [10], page 5), it is as if* `RESHAPE(R, SHAPE(Q))` *had been specified. 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.*

Opposed to its predecessor system, VFC also supports the declaration of local processor arrays within procedures.

3.2 Data Distribution Mechanisms

The major improvements of VFC concerning data distribution are the full support for the distribution of dynamically allocated objects including automatic arrays, dynamic data distribution and redistribution, and the possibility of parameterizing the distribution of an array by the number of processors as outlined above.

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-2 and the HPF-2 Approved Extensions. Opposed to VFCS which could handle only indirect distributions that were restricted to a single array dimension, VFC implies no restrictions on the use of `INDIRECT`. As in HPF, the number of distributed dimensions must always be equal to the rank of the corresponding processor array (section).

Example 3.2: Distributions provided by VFC

```
      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)
!HPF$ DISTRIBUTE(INDIRECT(MAP),*) :: A2
```

3.3 Runtime-defined Static Distributions

Opposed to VFCS, VFC supports run-time 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 3.3.

Example 3.3: 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

An array that should be dynamically allocated 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 intrinsic function `ASSOCIATED` may be used. Dynamic memory allocation for distributed objects is supported by VFC without any restrictions.

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

Example 3.4: 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 dynamically allocated 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 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.

Due to current restrictions of the runtime system, data redistribution is only supported for arrays that have a rank not greater than 3. Note also that the redistribution routines used currently in VFC's runtime system are not optimized and thus may cause a huge runtime overhead. 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.

Dynamic redistribution is also supported for arrays that are dynamically allocated. In such cases an array must have been associated with a distribution prior to allocation.

Example 3.5: 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, 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 dynamically distribute arrays. 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 Distribution of Dummy Arrays and Local Arrays

Dummy arguments of procedures may either inherit their distribution from the actual arguments using the INHERIT attribute or may have an explicit distribution declared for it. In the latter case

the distribution of the actual and formal array must match. If this cannot be verified by VFC a corresponding *restriction message* is issued. Note, that VFC currently requires that distributed dummy arguments are assumed shape arrays. Another current restriction is that dummy arrays with the DYNAMIC attribute must not have in addition the INHERIT or DISTRIBUTE attribute.

Local arrays of procedures (including automatic arrays) may be distributed in the same way as arrays of the main program unit.

Example 3.6: Distribution of Dummy Arrays and Local Arrays

```
!HPF$ PROCESSORS R(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE(BLOCK) ONTO R :: A, B, C, D
      REAL, DIMENSION(1000)    :: A, B, C, D, E

      CALL SUB(A, B, C, D, E(i))

      SUBROUTINE SUB(V, W, X, Y, Z)
      REAL, DIMENSION(:)      :: V, W, X
      REAL, DIMENSION(100,10) :: Y
      REAL, DIMENSION(*)      :: Z

!HPF$ PROCESSORS RL(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE(BLOCK)  ONTO RL :: V
!HPF$ INHERIT                    :: W
!HPF$ DISTRIBUTE(CYCLIC) ONTO RL :: X  ! mismatch of dummy and actual distr.
!HPF$ INHERIT            ONTO RL :: Y  ! not supported
```

This example shows the use of distributions specifications for dummy arguments. Distributed dummy arguments must always be assumed shape arrays. Furthermore, the distribution of a statically distributed dummy array must either be specified with INHERIT *or with the* DISTRIBUTE *attribute. The explicit distribution specification for* X *causes a restriction message since it does not match the distribution of the actual argument* C*. The use of* Y *causes a restriction message since the shape of* Y *does not conform with the shape of* D*. The assumed size declaration for* Z *is acceptable since neither* Z *nor* E *are distributed.*

3.7 The INDEPENDENT Directive

All HPF+ kernels include loops where distributed arrays are accessed by means of indirection arrays which are defined at run-time. All these loops do not contain loop carried dependences an 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 3.7 shows an `INDEPENDENT` directive with a `NEW` clause and a `REDUCTION` clause.

Example 3.7: `The INDEPENDENT Directive`

```
!HPF$ INDEPENDENT, NEW(nn1,nn2,nn3,nn4), REDUCTION(f)
      do iel = 1, numel2
        nn1 = ix(1,iel)
        nn2 = ix(2,iel)
        nn3 = ix(3,iel)
        nn4 = ix(4,iel)
        do idim=1,6
          f(idim,nn1) = f(idim,nn1) + force1(idim,iel)
          f(idim,nn2) = f(idim,nn2) + force2(idim,iel)
          f(idim,nn3) = f(idim,nn3) + force3(idim,iel)
          f(idim,nn4) = f(idim,nn4) + force4(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` and `ON` clause may be used. With these clauses the user may specify that a loop iteration should be executed by the processor that owns a particular array element or on a specific processor, respectively.

Example 3.8: `ON HOME` **and** `ON` **Clauses**

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

!HPF$ INDEPENDENT, ON R(4-MOD(J,4))
      DO J=1,N
         A(J) = ...
      END DO
```

In this example, iteration I of the first loop will be executed on the processor that owns array element `A(I)`*. The on-clause of the second loop specifies that iteration J of the second loop will be executed on processor* `R(4-MOD(J,4))`*.*

`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) or across different loops with similar access patterns. 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 providing 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 schedule of all arrays accessed in the loop is invariant for *all* executions of the loop and thus has 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 3.9: 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 (re)computed in iteration* $1, 11, 21, \ldots$*, of the outer loop.*

A specification of the syntax of `INDEPENDENT` directives and *reduction statements* as currently implemented in VFC can be found in the Appendix.

3.8 Pure and Purest Procedures

The `PUREST` directive enables the user to assert that a procedure may be executed without the need of communicating data between the processors. The `PUREST` directive has to be specified immediately after the procedure or function statement and may only be specified if the `PURE` attribute is present.

If a procedure that is called from an `INDEPENDENT` loop is purest, much more efficient code can be generated. The purest directive has been proven to be very useful for the HPF+ PAM CRASH kernels which call from `INDEPENDENT` loops procedures that exclusively operate on *private* data objects. This is shown in Example 3.10.

Example 3.10: PUREST **Procedures**

```
      real                       :: f(6,numnp),x(3,numnp),ix(4,numel2)
      real, dimension(6,numel2) :: force1,force2,force3,force4

!HPF$ DISTRIBUTE(*,BLOCK)        :: f,x,ix,force1,force2,force3,force4

      real xn(3,4),force(6,4)      ! private variables
      ...

!HPF$ INDEPENDENT, NEW(nn1,nn2,nn3,nn4,xn,force)
      DO iel = 1, numel2

         nn1 = ix(1,iel)
         nn2 = ix(2,iel)
         nn3 = ix(3,iel)
         nn4 = ix(4,iel)

         xn(:,1) = x(:,nn1)
         xn(:,2) = x(:,nn2)
         xn(:,3) = x(:,nn3)
         xn(:,4) = x(:,nn4)

         call mforce(force,xn)    ! call purest procedure

         force1(:,iel) = force(:,1)
         force2(:,iel) = force(:,2)
         force3(:,iel) = force(:,3)
         force4(:,iel) = force(:,4)

      END DO

      ...

      PURE SUBROUTINE mforce(force,x)
!HPF$ PUREST
           dimension force(6,4), x(3,4)
           ...
      END SUBROUTINE mforce
```

This example which is taken from the PAM CRASH kernel developed by NEC shows the use of PUREST *procedures. Before the call to the purest subroutine* mforce *data from the distributed array* x *is copied to the* NEW *(private) array* xn*. The subroutine* mforce *which performs the bulk of the computation works exclusively on private data and does not require any communication. After return from* mforce *data is copied back from the private array* force *to the distributed arrays* force1, ..., force4*.*

3.9 Extrinsic Interfaces

VFC supports the extrinsic interfaces `HPF_LOCAL` and `HPF_SERIAL` and the associated argument transfer mechanisms but not yet the `HPF_LOCAL` library. It is planned to implement at least the most basic functions of this library, for example, `MY_PROCESSOR`, `GLOBAL_SIZE` and `GLOBAL_SHAPE`.

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 the reader is referred to [19].

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

4 Compilation

Overview

In this section we describe the compilation techniques which have been implemented in VFC for the parallelization of HPF+ programs. VFC adopts a very general and flexible parallelization strategy which not only is applicable to HPF-2 but also to the dynamic concepts of the HPF-2 Approved Extensions. 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. The main focus within the HPF+ project is on the parallelization of independent loops, in the context of irregular, possibly dynamically changing, data distributions and access patterns. At the moment only limited support for an efficient parallelization of regular constructs is provided. Thus a large overhead may be introduced for programs that exhibit only regular distributions and access patterns. Efficient support also for highly regular constructs is planned for future releases of the system but is not considered as the main objective of this project.

In the following we given an overview of the processing of distribution directives, the management of distributed objects at compile time and at runtime, the allocation strategy for distributed arrays, the transformation of array assignment, the parallelization of 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 Processing of PROCESSORS Directives

For each processors arrangement 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: Processing of PROCESSORS Directives

HPF processors statement ...

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

generated Fortran 90 code ...

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

4.2 Management of Distributed Arrays

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.

VFC supports the use of the `NUMBER_OF_PROCESSORS` intrinsic function within processor array declarations. A distributed array that is mapped to a processor array whose size is not known at compile time cannot be allocated statically in the target program since the size of the local segment that has to be allocated on each processor cannot be determined by the compiler. To handle such cases, VFC currently adopts a very general strategy which transforms all declarations of distributed arrays in such a way that in the target program each processor allocates its local segment dynamically.

4.2.1 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 procedures has been strongly influenced by the need to support the dynamic concepts of HPF and in particular of its Approved Extensions.

A runtime layout descriptor consists of three main components, the memory descriptor, the distribution descriptor and the alignment descriptor. 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 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 pointer to the alignment-target array.

All layout descriptors for a program unit are managed using a dynamically created descriptor 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 mechanism is particularly important to implement sharing of descriptors in the context of dynamic data distribution. Furthermore, runtime tests to check whether two dynamically distributed arrays have the same distribution can be efficiently realized with this strategy.

4.2.2 Statically Distributed Arrays

Statically distributed arrays have a fixed distribution within their scope of declaration. Thus at the beginning of the executable section of the corresponding program unit the distribution specifications are evaluated and the results are stored in the layout descriptor. Then the array is dynamically allocated according to its local size from the layout descriptor. 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.

Processing of statically distributed arrays consists of the following steps:

1. Generation of a pointer to a *layout descriptor*
2. Generation of call(s) to initialize the layout descriptor
3. Generation of `ALLOCATE` statement for local segment
4. Transformation of array declaration
5. Generation of `DEALLOCATE` statement for local segment

Example 4.2: Processing of Statically Distributed Arrays

declaration of statically distributed array ...

```
      REAL A(N)
!HPF$ PROCESSORS R(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE(INDIRECT(MAP)) ONTO R :: A
```

generated Fortran 90 code ...

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

      ! initialize layout descriptor
      CALL vhpf_init_rt_dsc(dlist, a_dsc, 2, shape=(/n/), dtype=(/INDIRECT/), &
                            aarg1=(/map/), proc=r_dsc)
      ! allocate local segment
      ALLOCATE((a(1:vhpf_extent(a_dsc, 1)))
      ...
      ...
      DEALLOCATE(a)
      CALL vhpf_free_rt_dsc(dlist, a_dsc)    ! free layout descriptor
```

This example shows the code generated for a statically allocated and statically distributed array. A pointer to a runtime layout descriptor is generated and the declaration of the array is changed such that it can be dynamically allocated. At the beginning of the executable section a call to the runtime procedure `vhpf_init_rt_dsc` *is generated which initializes the array's layout descriptor. After this call the local segment (including buffer space for non-local elements) is allocated on each processor according to the computed size which is stored in the descriptor and returned by the function* `vhpf_extent()`*. At the end of the executable section of the corresponding program unit the local memory and the layout descriptor for the distributed array are deallocated.*

Depending on the distribution scheme (static or dynamic) and on the allocation scheme of an array the initialization of layout descriptors may have to be performed incrementally in separate steps as outlined in the following sections.

4.2.3 Dynamically Allocated Distributed Arrays

The initialization of layout descriptors for dynamically allocated arrays[3] is done in several separate steps. First the initialization routine `vhpf_init_rt_dsc` is generated with the distribution information of the array only (if present) and the corresponding descriptor is partially initialized. The allocation of the array is delayed until an `ALLOCATE` statement is executed.

Transformation of `ALLOCATE` Statements

Prior to each `ALLOCATE` statement for a distributed array a call to the library routine `vhpf_init_rt_dsc` is generated taking as argument only the shape information from the `ALLOCATE` statement resulting in a complete initialization of the layout descriptor. Then the `ALLOCATE` statement is transformed such that on each processor only the local segment is allocated according to the size obtained from the layout descriptor.

Example 4.3: Processing of Dynamically Allocated Distributed Arrays

declaration of a dynamically allocated distributed array ...

```
      REAL, ALLOCATABLE :: B(:)
!HPF$ PROCESSORS R(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE(CYCLIC) ONTO R :: B
      ...
      ALLOCATE(B(N), STAT=ALLOC_ERR)
```

generated Fortran 90 code ...

```
      REAL, POINTER, DIMENSION(:) :: b
      TYPE (rt_dsc), POINTER :: b_dsc
      ...

      ! initialize descriptor with distribution info only
      CALL vhpf_init_rt_dsc(dlist, b_dsc, 2, dtype=(/CYCLIC/), proc=r_dsc)
      ...
      ! update descriptor with shape info
      CALL vhpf_init_rt_dsc(dlist, b_dsc, 2, shape=(/N/),  proc=r_dsc)

      ! allocate local segment
      ALLOCATE((b(1:vhpf_extent(b_dsc, 1)), STAT=ALLOC_ERR)
      ...
```

This example shows the transformation of `ALLOCATE` *statements. At the beginning of the executable section a call to* `vhpf_init_rt_dsc` *is generated 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` *statement is transformed such that instead of the global size only the local segment is allocated on each processor.*

[3] These are arrays that have the `ALLOCATABLE` or `POINTER` attribute.

4.2.4 Dynamically Distributed Arrays

If an array with a DYNAMIC attribute does not have an initial distribution specified for it, then the layout descriptor initialization routine is called without the distribution information and the descriptor is only partially initialized. As a consequence, the allocation of such arrays has to be delayed until the array has been associated with a distribution, which is usually done by executing a REDISTRIBUTE statement.

Transformation of REDISTRIBUTE Statements

A REDISTRIBUTE statement is transformed as follows. First a temporary layout descriptor for the array is generated. Then, prior to the REDISTRIBUTE statement, a call to initialize the temporary descriptor with the distribution information from the REDISTRIBUTE statement is generated. After that, a call to the runtime routine vhpf_redist, which takes as arguments both the original and the temporary layout descriptor, is generated. Within this redistribution routine 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.

Example 4.4: Processing of Dynamically Distributed Arrays

declaration of a dynamically distributed array ...

```
      REAL, DIMENSION(N) :: C
!HPF$ PROCESSORS R(NUMBER_OF_PROCESSORS())
!HPF$ DYNAMIC :: C
      ...
!HPF$ REDISTRIBUTE(BLOCK) ONTO R :: C
```

generated Fortran 90 code ...

```
      REAL, POINTER, DIMENSION(:) :: c
      TYPE (rt_dsc), POINTER :: c_dsc
      TYPE (rt_dsc), POINTER :: c_dsc_tmp  ! temporary layout descriptor
      ...
      ! initialize descriptor with shape info only
      CALL vhpf_init_rt_dsc(dlist, c_dsc, 2, shape=(/N/), proc=r_dsc)
      ...
      ! initialize temporary descriptor with new distribution info
      CALL vhpf_init_rt_dsc(dlist, c_dsc_tmp, 2, dtype=(/BLOCK/), proc=r_dsc)

      ! call redistribution routine
      CALL vhpf_redist(dlist, c, c_dsc, c_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 a runtime routine to perform the actual redistribution is inserted. Within the routine* vhpf_redist *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.*

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 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.

4.2.5 Discussion

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

However, experiments have shown that with some Fortran 90 compilers 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. For this reason, it is planned to avoid in future versions of VFC the use of `POINTER` attribute for distributed objects in the target program whenever possible.

4.3 Processing of Function Calls

Calls to *array intrinsic functions* with distributed arguments are substituted by calls to corresponding versions of these functions from the runtime 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 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 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.

4.4 Processing 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 mapped actual argument is associated with a non-mapped 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

```
!HPF$ PROCESSORS R(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE(BLOCK) ONTO R :: A, B
      REAL, DIMENSION(N) :: 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)
```

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 the distributed argument its layout descriptor.*

If a dummy array has the INHERIT attribute it may be associated with an actual array with any distribution. If a dummy array is explicitly distributed (i.e., 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 distribution.

The interface of a procedure with distributed arguments[4] is transformed such that for every distributed argument an additional dummy argument to pass the runtime layout descriptor is generated.

In cases where a mapped actual argument is associated with a non-mapped 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 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 of this potential runtime overhead by corresponding *diagnostic messages*.

Example 4.7: Redistribution at Procedure Boundaries

```
!HPF$ DISTRIBUTE(BLOCK) ONTO R :: A
      REAL, DIMENSION(N) :: A
      ...
      INTERFACE
         SUBROUTINE SUB(X)
            REAL, DIMENSION(:) :: X
         END SUBROUTINE SUB
      END INTERFACE
      ...
      CALL  SUB(A)
```

generated code ...

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

In this example a distributed actual array is passed to a non-distributed formal array. This is handled by replicating the array before the call and restoring the original distribution after the call.

4.5 Processing of I/O Statements

VFC currently provides only limited support for the I/O of distributed arrays. Either whole distributed arrays or single elements thereof, but not array sections, may be read in. I/O statements 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 sends the data read to the 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.

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

[4] Currently a dummy argument may have only one of the attributes DISTRIBUTE, INHERIT, or DYNAMIC.

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 4.8: Transformation of I/O Statements

```
      INTEGER A(M,N), B(M,N), C(N), X(10), S

!HPF$ PROCESSORS, DIMENSION(NUMBER_OF_PROCESSORS()) :: R
!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.*

4.6 Processing of Scalar Assignments Outside Independent loops

Scalar assignments to distributed arrays which do not occur within independent loops are transformed in such a way that they are executed on the processor that owns this array element. If in addition on

the right-hand-side of the assignment array elements that might be located on a different processor are referenced, then communication statements have to be generated in front of the assignment statement in order to communicate the array element from their owners into temporary variables on the executing processor. The assignment statement is 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`.

Example 4.9: Worst-case Parallelization Strategy

```
!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(I)` *from the processor that owns it as a result of executing the runtime 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.

4.7 Parallelization of Array Assignments

This version of VFC parallelizes only a restricted class of array assignment statements that operate on arrays with the same distribution and which do not require the transfer of data. If the system can proof that no communication will be required it transforms array assignment statements in such

a way that each processor only accesses data local to it. Note that VFC currently requires that within array assignment statements accesses to a distributed array dimension do not refer to sub-sections.

More precisely, array assignment statements referencing mapped arrays are currently handled only if the following restriction (which is checked by the system) is satisfied.

- The left hand side is a *whole* array (e.g. A) or an array section where distributed dimensions are accessed as a whole and the right hand side expression is either
  - a scalar expression which does not reference sections of mapped arrays,
  - or an array expression where all referenced non-scalar objects are whole arrays or array sections where distributed dimensions are accessed as a whole and each distributed dimension on the left-hand-side is distributed in exactly the same way as the corresponding dimension of A.

A general parallelization strategy that is applicable to a much broader class of array assignments is currently being implemented.

Example 4.10: Transformation of Array Assignments

```
!HPF$ PROCESSORS R(4,4)
!HPF$ DISTRIBUTE(BLOCK,CYCLIC)   :: A
!HPF$ DISTRIBUTE(BLOCK,*,CYCLIC) :: B

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

! original array assignment
B(:,I,:) = A(:,:)
```

transformed array assignment ...

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

This example shows the current parallelization of array assignments that do not require communication. The runtime routine `vhpf_loc_ub` *returns the upper bound of the local segment (excluding potential buffer space).*

4.8 Parallelization of Independent Loops

The compilation techniques applied for parallelizing independent loops differ significantly from those applied to other parts of the program[5]. 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

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

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 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.`.

Despite these major advances there are still some restrictions for independent loops which will be relaxed in future versions of VFC. If any of the current restrictions is violated the loop will be sequentialized and transformed based on the single-element strategy. If this is the case the user is informed by corresponding *restriction messages*. The current restrictions are summarized in Section 6.2.

4.8.1 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 independent 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 phase can be described as the *analysis phase* and the next ones as *code generation phases*. The analysis phase traverses the program, constructs internal data structures, and analyzes the structure of independent loops. 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.

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 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.

4.8.2 Independent 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 the given independent loop nest is represented by a *list-structure*, where each loop inside the independent loop nest is condensed to a single item in the list (see Fig. 1).

```
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;       /* seqpar = 0  if independent         *
                                      * seqpar > 0  others                 */
    PTypeIdentEntry    loop_entry;   /* pointer to loop iteration var      */
    tTree              loop_init;    /* pointer to loop init bound in AST */
} DOIND_NODE;
```

Figure 1: Independent loop nest descriptor

Associated with each independent loop nest is a list of descriptors of all arrays occurring in the loop nest. Every array descriptor (see Fig. 2) summarizes information about characteristics of the array and records all its references.

The independent loop nest descriptor is constructed and partially initialized by traversing the loop nest. These initial settings are updated in subsequent analysis phases. For each array reference an *offset number* and *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 an array reference the corresponding loop. It is represented as n-tuple of integer numbers, where n is the rank of the array; each

```
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            */
} 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               reduct_op;      /* reduction operator               */
    int               def_comm;       /* optimization flag for scatter    */
    int               use_comm;       /* optimization flag for gather     */
    int               use_eq_def;     /* optimization flag                */
} DOIND_ARR_NODE;
```

Figure 2: Array descriptor

component denotes the sequence number of the corresponding loop. It is not allowed that a subscript expression of a dimension is dependent on more than one loop. For example, $A(i+j)$, where i and j are loop iteration variables of two different loops, is illegal.

Checking for locality of references is performed by comparing each array reference symbolically with the on-clause on a dimensional basis. If all references of an array are found to be local, no calls to communication procedure need to be generated.

4.8.3 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 Fig. 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 execution space of an independent loop nest in the executing processor. The size of `EXEC` is equal to the nesting depth n of the loop nest, that is `EXEC(i)` stores the execution space of the loop with nesting level $i-1$, $1 \leq i \leq n$. `EXEC(1)` always contains the execution space for the outermost independent loop. Local iterations to be performed on the executing processor, are stored in the component `ITERS`, and are determined by the work distributor.

For each distributed array X referenced in an independent loop the following auxiliary data structures are generated (see also Fig. 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
              INTEGER, DIMENSION(N1) :: x
              INTEGER, DIMENSION(N2) :: y
              ...
      !HPF$   PROCESSORS :: PP1(NP*NP), PP2(NP, NP)
      !HPF$   DISTRIBUTE(BLOCK, BLOCK) ONTO PP2 :: a, b
      !HPF$   DISTRIBUTE(BLOCK) ONTO PP1        :: x
              ...
      !HPF$   INDEPENDENT, ON HOME(a(:,i))
(L1)          DO i= l1, u1
      !HPF$     INDEPENDENT, ON HOME(a(j,:))
(L2)            DO j= l2, u2
                  a(j, i) = b(i,j) + b(x(i),y(j))
                END DO
              END DO
```

Figure 4: Example of independent loops

4.8.4 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 dependend 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 clause specified in the INDEPENDENT directive. If no work distribution specification is present, it is derived automatically by the compiler.

Usually work distribution is implemented by a call to the library routine VHPF_BUILD_EXEC_OW, which determines the execution set on the executing processor according to the ownership of a certain distributed array dimension. The dimensional execution set for a sequential loop inside the independent loop nest is equal to the whole iteration set.

Consider the independent loop nest of Fig. 4, where A, B, and X are distributed arrays, and Y is replicated. The corresponding work distributor code together with auxiliary variable declarations

is shown in Fig. 5. The routine VHPF_BUILD_EXEC_OWO implements an optimized version of the work distribution algorithm which is utilized whenever subscripts are equal to 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_OWO(a_dsc, l1, u1, 2, 1, exec)

! -<< Dimensional work distribution for loop (L2) ==> EXEC(2) >>-
      CALL VHPF_BUILD_EXEC_OWO(a_dsc, l2, u2, 1, 2, exec)

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

Figure 5: Work distributor code

4.8.5 Inspector

The inspector performs runtime analysis of the independent loop nest. It generates a description of the communication necessary for the loop nest in the form of *schedules*. Schedules control runtime procedures moving 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 accessed 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 a 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 corresponding *local reference lists* stored in 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 that will contain 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 will be re-allocated with a larger buffer. If the compiler is able to detect that only the local elements of an array are accessed, no communication procedure calls

```
! -<< Collect 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_gbox%offset => x_offset
       x_lbox%noccur =  1
       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)

! -<< Collect global subscripts of A >>-
       offs = 0
       a_offset(1,1) = offs         ! 1st reference, 1st dimension
       DO ni = 1, exec(2)%len
         a_gbox%dim(1)%indx(ni+offs) = exec(2)%iters(ni)
       END DO
       offs = offs + exec(2)%len
       a_offset(1,2) = offs
       offs = 0
       a_offset(2,1) = offs         ! 1st reference, 2nd dimension
       DO ni = 1, exec(1)%len
         a_gbox%dim(2)%indx(ni+offs) = exec(1)%iters(ni)
       END DO
       offs = offs + exec(1)%len
       a_offset(2,2) = offs
       a_gbox%noccur =  1
       a_gbox%offset => a_offset
       a_lbox%noccur =  1
       a_lbox%offset =  a_offset
! -<< Global to local index conversion; no communication is needed >>-
       CALL PARS_AGGREGATE_G2L(a_dsc, a_gbox, a_lbox)
```

Figure 6: Inspector code (part 1)

```
! -<< Collect 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
       b_offset(2,3) = offs
       b_gbox%noccur =  2
       b_gbox%offset => b_offset
       b_lbox%noccur =  2
       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)
```

Figure 6: Inspector code (part 2)

are generated. In this case the runtime procedure PARS_AGGREGATE_G2L is utilized for global to local index conversion.

Consider the independent loop nest given in Fig. 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 Fig. 6. Array accesses are processed in multi-phases starting with the innermost subscripts. In one phase only array references with the same indirection level are resolved. For an array X wich is used as an indirection array of another distributed array A non-local elements must be fetched before using it to reference A.

4.8.6 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.

```
! -<< 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 >>-
       a_tindx1 => a_lbox%dim(1)%indx
       a_tindx2 => a_lbox%dim(2)%indx
       a_toffs1_01 = a_lbox%offset(1,1)
       a_toffs2_01 = a_lbox%offset(2,1)
       ...
!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(a_tindx1(j + a_toffs1_01), a_tindx2(i + a_toffs2_01)) = &
     &     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))
         END DO
       END DO
```

Figure 7: Executor code

The executor code for the loop nest of Fig. 4 is shown in Fig. 7. For moving data between processors the library procedures VHPF_GATHER and VHPF_SCATTER are utilized. To increase the performance of the generated executor loop auxiliary data structures are used to store invariant variables used in addressing array elements.

4.8.7 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. *Initializing* the scalar variable or buffer space of the distributed array.
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.

```
        REAL, DIMENSION(N1,N2) :: a, b
        INTEGER, DIMENSION(N1) :: x
        INTEGER, DIMENSION(N2) :: y
        REAL                   :: r
        ...
!HPF$   PROCESSORS :: PP1(NP*NP), PP2(NP, NP)
!HPF$   DISTRIBUTE(BLOCK, BLOCK) ONTO PP2 :: a, b
!HPF$   DISTRIBUTE(BLOCK) ONTO PP1        :: x, y
        ...

!HPF$   INDEPENDENT, ON HOME(x(i)), REDUCTION(a, r)
        DO i= l1, u1
!HPF$     INDEPENDENT, ON HOME(y(j))
          DO j= l2, u2
            a(x(i),y(j)) = a(x(i),y(j)) + b(x(i),y(j))
            r = r + b(x(i),y(j))
          END DO
        END DO
```

Figure 8: Example of independent loops with REDUCTION clause

Consider the independent loop nest shown in Fig. 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 Fig. 9.

```
! -<< Initialization >>-
        s = r; r = 0
        CALL VHPF_RSET_BUF(a_dsc, 0, lseg_2d=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 >>-
        a_tindx1 => a_lbox%dim(1)%indx
        a_tindx2 => a_lbox%dim(2)%indx
        a_toffs1_01 = a_lbox%offset(1,1)
        a_toffs2_01 = a_lbox%offset(2,1)
        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)
!HPF$   INDEPENDENT, REDUCTION (a, r), ON HOME (x(i))
          DO i = 1, exec(1)%len
!HPF$       INDEPENDENT, ON HOME (y(j))
              DO j = 1, exec(2)%len
                a(a_tindx1(i + a_toffs1_01), a_tindx_2(j + a_toffs2_01)) = &
     &          a(a_tindx1(i + a_toffs1_01), a_tindx_2(j + a_toffs2_01)) + &
     &          b(b_tindx1(i + b_toffs1_01), b_tindx_2(j + b_toffs2_01))
                r = r +  &
     &              b(b_tindx1(i + b_toffs1_01), b_tindx2(j + b_toffs2_01))
              END DO
          END DO

! -<< Add non-local elements of A to their original values >>-
        CALL VHPF_RSCATTER_ADD(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

4.8.8 Optimizations and Communication Schedule Reuse

The goal of the optimization phase is to reduce the cost of runtime preprocessing performed in the inspector, which usually introduces a large overhead. 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

```
           LOGICAL, SAVE :: lfirst1 = .TRUE., lfirst2 = .TRUE.
           ...
    ! -<< Loop (L1) >>-
           IF (lfirst1) THEN
    !        -<< Work distributor >>-
    !        -<< Inspector >>-
             lfirst1 = .FALSE.
           END IF
    ! -<< Executor >>-
    ! -<< Loop (L2) >>-
           IF (lfirst2 .OR. .NOT.rcond) THEN
    !        -<< Work distributor >>-
    !        -<< Inspector >>-
             lfirst2 = .FALSE.
           END IF
    ! -<< Executor >>-
```

Figure 11: Implementation of the 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 Fig. 11 for the independent loops of Fig. 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.

4.9 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. This is shown in Example 4.11.

Example 4.11: Processing of Calls to Extrinsic Functions

```
INTEGER, DIMENSION(100) :: A

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

      INTERFACE
        EXTRINSIC(HPF_LOCAL) INTEGER FUNCTION LOCFUN(X)
          INTEGER, DIMENSION(:) :: X
        END FUNCTION LOCFUN

        EXTRINSIC(HPF_SERIAL) INTEGER FUNCTION SERFUN(X)
          INTEGER, DIMENSION(:) :: X
        END FUNCTION SERFUN
      END INTERFACE

      I = LOCFUN(A)
      J = SERFUN(A)
```

generated code ...

```
      ...
      x = locfun(a(1:vhpf_loc_ub(a_dsc, 1):1))   ! call hpf-local function

      CALL vhpf_undist(a, dlist, a_dsc)          ! replicate A
      x = serfun(a)                              ! call hpf-serial function
      CALL vhpf_restore(a, dlist, a_dsc)         ! restore original distribution
```

The call to LOCFUN *is transformed such that on each processor only the local section of the distributed array* A *is passed. The call to* SERFUN *is transformed by inserting a call to* vhpf_undist *which replicates* A *on each processor before the invocation of* SERFUN*, and a call to* vhpf_restore *which restores the original distribution after the call.*

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 runtime support for dynamic data redistribution. It provides high level interfaces for the PARS library calls which significantly alleviate code generation. The PARS library provides low level communication routines uitilizing 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. 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 distribution 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 distribution descriptors is searched in order to find a matching distribution descriptor, and if one is found a pointer to it is returned. If no matching descriptor is found, a new layout 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 distribution descriptors in program unit.
  - Actual layout descriptor.
  - Temporary layout descriptor.

- **Output**
  - Redistributed array.
  - Updated layout descriptor.

- **Description**
  This function redistributes an array according to the new distribution of the temporary 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 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 distribution descriptors in program unit.
  - Actual layout descriptor.

- **Output**

  - Replicated array.

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

vhpf_restore() : Subroutine

- **Input**
  - Replicated array.
  - List of existing distribution descriptors in program unit.
  - Layout descriptor.
- **Output**
  - Distributed Array.
- **Description**
  This function distributes a replicated array according to the distribution of the layout descriptor.

5.2 PARS Runtime Library

PARS (PArallel Runtime System) is a successor of the PARTI [13] 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 references to the data array on each processor.
- **Output**
  - Equivalent local references of the global indices on each processor.
  - 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 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.

5.2.4 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.

6 Using VFC

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`.

6.1 Command Line Syntax

The VFC command-line syntax is

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

Where:

options is zero or more command-line options (see below).

filename is the name of a source file. More than one filename may be specified.

Options is zero or more of:

`-f` free source form

`-i` *path* Add *path* to the list of directories in which to search for INCLUDE files.

`-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 file.
Name the final executable file *out* and the intermediate Fortran 90 message passing program *out*`_vfc.f90` instead of `a.out` and `a_vfc.f90`.

`-s` Stop after generation of the intermediate message passing F90 code.

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.

6.2 Restrictions

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

- Data distribution
  - The distribution of objects of type `CHARACTER`, `LOGICAL` and `COMPLEX` is not yet supported.
  - The distribution of objects of derived types and components of derived types is not yet supported.
  - The specification of mappings by means of alignment is not yet supported.
  - *Prescriptive*, *descriptive* and *transcriptive* mappings for dummy arguments are not yet supported.
  - The HPF intrinsic and library procedures are not yet supported, with the exception of `NUMBER_OF_PROCESSORS()`.

- `INDEPENDENT` loops

  The parallelization of `INDEPENDENT` loops is currently subject to the following restrictions:
  - No subscript of a distributed array referenced within an `INDEPENDENT` loop may depended on more than one loop iteration variable.
  - The bounds-expressions of loops that are nested within `INDEPENDENT` loops must not reference outer loop variables.
  - Mapped (distributed) objects must not be referenced in the bounds of loops that are nested within `INDEPENDENT` loops.
  - Mapped objects must not be referenced within the body of conditional statements within `INDEPENDENT` loops.
  - `INDEPENDENT` loops and any loops that are nested within an `INDEPENDENT` loop must not have a step-size other than 1.
  - Procedures called from within an `INDEPENDENT` loop must be `PUREST`.
  - The `SELECT`, `REDISTRIBUTE`, `ALLOCATE` and `DEALLOCATE` statements are not yet supported within `INDEPENDENT` loops.

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

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

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

7 Conclusion and Future Work

In this document we described the HPF/HPF+ compilation system VFC which has been developed within the HPF+ project. Compared to its predecessor VFCS, VFC is fully based on 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, 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 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.

Despite these major advances, the parallelization of independent loops still imposes a number of restrictions. For example, it is currently not possible to access distributed arrays within conditionals in independent loops. This restriction and others will be removed in the next version of VFC.

VFC currently provides only limited support for an efficient parallelization of regular constructs. Therefore, a number of regular constructs are currently processed with runtime parallelization techniques and as a consequence the efficiency of such codes may be very poor. Moreover, in order to handle the dynamic concepts of HPF like dynamic data redistribution it was necessary to develop a very general parallelization strategy. This introduces additional overheads for codes that are inherently static. Despite the fact that optimization techniques for regular codes is not the main focus of this project we are planning to implement also the most important optimizations for regular codes.

The following extensions are planned for the next release of VFC:

- The current restrictions on independent loops will be relaxed, in particular the constraint on the use of conditional statements and the invariance of loop bounds.
- Analysis and optimization techniques for independent loops will be implemented. The main goal is to reduce whenever possible the overheads of indirect accesses which currently may be introduced in the executor code also for originally direct array references.
- Optimizations in the context of the allocation scheme adopted for distributed arrays will be performed. The use of `POINTER` attributes will be avoided whenever possible.
- At least simple alignments will be implemented.
- Support for the `RESIDENT`, `SHADOW` and `HALO` will be realized.
- Certain optimizations of the runtime library will be performed.

References

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

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

[3] 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

[4] 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.

[5] 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.

[6] 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.

[7] 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, Springerg.

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

[9] S. Benkner, H. Zima. HPF+ Initial Language Definition. Deliverable D2.1a. University of Vienna, June 1996.

[10] S. Benkner, H. Zima. Definition of HPF+ Release 2. Deliverable D2.1b. University of Vienna, February 1997.

[11] S. Benkner, P. Mehrotra, J. Van Rosendale, H. Zima. High-Level Management of Communication Schedules in HPF-like Languages, Technical Report 97-5, University of Vienna, Institute for Software Technology and Parallel Systems, April 1997.

[12] 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.

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

[14] 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.

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

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

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

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

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

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

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

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

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

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

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

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

[27] 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	*inherit-directive*
	or	*dynamic-directive*
	or	*combined-directive*
executable-directive	**is**	*redistribute-directive*
	or	*independent-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 *declaration-construct* may appear.

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

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

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

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

A.2 The PROCESSORS Directive

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

A.3 The DISTRIBUTE Directive

distribute-directive	**is**	`DISTRIBUTE` *distributee dist-directive-stuff*
	or	`DISTRIBUTE` *dist-attribute-stuff* `::` *distributee-list*
dist-directive-stuff	**is**	*dist-format-clause* [*dist-onto-clause*]
dist-attribute-stuff	**is**	*dist-directive-stuff*
	or	*dist-onto-clause*
distributee	**is**	*object-name*
dist-format-clause	**is**	`(` *dist-format-list* `)`
dist-format	**is**	`BLOCK` [`(` *int-expr* `)`]
	or	`CYCLIC` [`(` *int-expr* `)`]
	or	`GEN_BLOCK (` *int-array-expr* `)`
	or	`INDIRECT (` *int-array-expr* `)`
	or	`*`
dist-onto-clause	**is**	`ONTO` *dist-target*
dist-target	**is**	*processors-name*
	or	*processors-section*
processors-section	**is**	*processors-name* `(` *section-subscript-list* `)`

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

Constraint: If a *dist-format-list* is specified, its length must equal the rank of each *distributee*.

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

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

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

A.4 The INHERIT Directive

inherit-directive	**is**	`INHERIT` *dummy-argument-name-list*

Constraint: Only dummy arguments may have the `INHERIT` attribute.

Constraint: An object that has the `INHERIT` attribute must not have the `DISTRIBUTED` attribute or the `DYNAMIC` attribute.

A.5 The INDEPENDENT Directive

independent-directive **is** `INDEPENDENT` [*independent-clause-list*]

independent-clause **is** *new-clause*
or *reduction-clause*
or *on-clause*
or *reuse-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` (*array-element*)
or `ON` (*processor-array-element*)

reuse-clause **is** `REUSE` [(*scalar-logical-expr*)]

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.

A.6 Reduction Statements

reduction-stmt **is** *reduction-var-ref* `=` *expr reduction-op reduction-var-ref*
or *reduction-var-ref* `=` *reduction-var-ref reduction-op expr*
or *reduction-var-ref* `=` *reduction-function*(*reduction-var-ref*, *expr*)
or *reduction-var-ref* `=` *reduction-function*(*expr*, *reduction-var-ref*)

reduction-var-ref **is** *variable*

reduction-op **is** `+`
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.7 The DYNAMIC Directive

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

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

Constraint: An object with the `SAVE` attribute may not be declared `DYNAMIC`.

A.8 The REDISTRIBUTE Directive

redistribute-directive **is** `REDISTRIBUTE` *distributee dist-directive-stuff*
or `REDISTRIBUTE` *dist-attribute-stuff* `::` *distributee-list*

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

A.9 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.