Extension of VFCS for the Compilation of Project Benchmarks Vers. 1

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

Deliverable:	D2.2a
Date:	December 1996
Status:	draft
Confidentiality:	confidential
Responsible:	University of Vienna

Revised: January 7, 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

Contents

1 Introduction

In this report we describe the Vienna Fortran Compilation System with a special focus on the HPF and HPF+ language extensions and the corresponding implementation techniques that have been realized in the first year of the HPF+ project.

The current version of VFCS, VFCS 1.3.1, is a major extension of VFCS 1.2, the version which was available at the start of the project. Whereas the input languages of VFCS 1.2 were mostly based on Vienna Fortran language features, VFCS 1.3 now also supports many HPF-1 and HPF-2 data distribution mechanisms, including block distributions, general block distributions, indirect distributions and distribution to processor subsets. The HPF-2 `INDEPENDENT` directive with an extension for the specification of work distribution has been implemented. Moreover, a prototype implementation of pure and purest procedures has been carried out.

Efficient techniques for handling loops with irregular access patterns have shown to be crucial for the HPF+ benchmark kernels. The preliminary mechanisms for compiling Vienna Fortran `FORALL` loops have been extended and generalized for the parallelization of `INDEPENDENT` loops with indirect data accesses. These techniques have been integrated with the existing *overlap parallelization strategy*. In this context the runtime support library of VFCS for irregular computations had to be extended and ported to MPI in order to run on the Meiko CS-2.

With the extended version of VFCS all four initial benchmark kernels have been successfully parallelized. First performance results and comparisons with commercial HPF-1 compilers indicate that the proposed HPF+ language extensions and the related implementation techniques are essential in order to be able to handle irregular applications like those considered within the HPF+ project with reasonable efficiency. However, additional language support, in particular for the optimization and reuse of communication schedules seems to be necessary.

The rest of this report is structured as follows. Section 2 gives an overview of VFCS. Section 3 describes the HPF and HPF+ language features that have been implemented. Section 4 describes the compilation techniques of VFCS with a special focus on techniques that have been implemented for the parallelization of irregular applications. Section 5 describes how VFCS may be used to either automatically or interactively parallelize and execute HPF programs. Section 6 summarizes the major achievements and outlines the future work planned for the second year of the HPF+ project.

2 Overview of VFCS

In this chapter we give an overview of the overall features of the Vienna Fortran Compilation System (VFCS), and outline the structure of the system and the major compilation techniques.

2.1 Major Aspects of VFCS

VFCS performs a source-to-source translation from data-parallel Fortran programs to explicitly parallel Message Passing Fortran, providing automatic parallelization as well as vectorization. VFCS supports the parallelization of sequential programs by providing integrated support for program analysis, profiling, performance prediction and performance measurement. Moreover, the system can be used to analyze and transform sequential programs. Many of the transformations which are used to improve the performance of parallel programs are useful for performance tuning of sequential code.

VFCS provides different modes of operation to the user. It can be used in an automatic mode, i.e. as a *command-line compiler*, or in an *interactive mode*. When used as a command-line compiler, VFCS provides, in addition to command-line options, a set of directives for the creation of *driver files* which control the sequence of transformations applied during the parallelization process. In interactive mode, VFCS offers a set of analysis services and a transformation catalog which allow the user to guide the system through the compilation process. This mode is supported by a hierarchically structured user interface based on X11-MOTIF.

Input languages for VFCS are Fortran 77 with Fortran 90 extensions, a subset of Vienna Fortran [24] and Vienna Fortran 90 [2], and a subset of HPF [18] with HPF+ [7] extensions. VFCS generates output for various distributed memory architectures including the Intel iPSC-860 Hypercube, the Intel Paragon, the Meiko CS-2 and networks of workstations using the PARMACS, NX, or MPI message passing libraries.

The parallelization techniques of VFCS are based upon the *Single-Program-Multiple-Data (SPMD)* paradigm. Currently two major parallelization techniques have been realized, the *Overlap* strategy [15, 16] and the *Inspector-Executor* strategy [19].

The overlap strategy is targeted towards regular computations, like stencil computations, and relies heavily on compile-time analysis and compile-time optimization.

The second parallelization strategy, the inspector-executor strategy, is targeted towards irregular computations, which are characterized by loops exhibiting irregular access patterns (using indirection arrays) which are dependent on runtime data. Thus, runtime preprocessing must be applied for such loops.

In the context of the HPF+ project special emphasis has been paid on the implementation and optimization of the inspector-executor strategy and its integration with the overlap strategy. A detailed description of the inspector-executor strategy and its implementation within VFCS can be found in Section 4.6.

2.2 Data Parallel Language Features

VFCS supports many of the Vienna Fortran and HPF-1 data parallel extensions. Moreover, within the HPF+ Project a number of language features of HPF-2, the HPF-2 Approved Extensions and of HPF+ have been implemented. In this report we limit our discussion to HPF and HPF+ distribution features and do not discuss the available Vienna Fortran language extensions. A detailed description of the implemented HPF and HPF+ language features can be found in Section 3 and in the Appendix.

Data Distribution Features

VFCS supports several variants of BLOCK distributions, replication, partial replication, distribution to processor subsets, distribution by alignment, and inherited distributions in the context of procedures. Currently VFCS provides the following data distribution features:

- *Regular block distributions*
- *General block distributions*
- *Indirect distributions*
- *Replication*
- *Distribution to processor subsets*
- *Multidimensional distributions*
- *Distribution by alignment*

Independent Loops

The HPF-1 `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 VFCS also supports the HPF-1 `NEW` clause for defining variables that are private to a loop iteration, the HPF-2 `REDUCTION` clause, and the HPF+ `ON` and `ON HOME` clauses which enable the user to specify how the loop iterations should be distributed among the processors. A detailed description of `INDEPENDENT` loops as currently implemented in VFCS is given in Section 3.4.

Purest Procedures

A *purest* procedure is a *pure* procedure whose invocation does not result in communication on entry, during activation, and upon exit from the procedure. In the current version of VFCS purest procedures which may be called from `INDEPENDENT` loops and which operate exclusively on data that is private to a processor have been implemented.

2.3 Sources of Parallelism

VFCS exploits the parallelism inherent in the following program constructs:

- `DO` loops:
  VFCS performs data dependence analysis to determine which loops of the program are parallelizable. If dependence analysis reveals that a loop is parallelizable, the loop is transformed such that the iterations are distributed to the available processors and appropriate communication statements are inserted before the loop to communicate non-local data.

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

- Reduction operations
  VFCS provides pattern matching techniques to determine whether a loop performs a reduction operation such as summing all elements of a vector, computing the dot-product of two vectors, computing the minimum or maximum value of a vector and others. Furthermore, VFCS handles the HPF-2 `REDUCTION` clause in the context of `INDEPENDENT` loops.

- `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 irregular `INDEPENDENT` loops is implemented based on run-time compilation techniques.

2.4 Overview of the Parallelization Strategy

SPMD Model

The parallelization strategy of VFCS is based on the *SPMD* programming model. VFCS translates a source program into an SPMD message passing program, in the following referred to as *target program.* Each processor of the target machine will execute the same target program, but on different data. VFCS parameterizes the target program such that, according to the data distribution, each processor operates on its local portion of the data.

Data Distribution Analysis

By analyzing the distribution specifications VFCS determines for each distributed array of the program the set of local elements that have to be allocated on each processor[1]. *Data distribution analysis* is performed on an interprocedural basis, i.e. by propagating data distribution information over the

[1] In case of `INDIRECT` distributions, which may be defined using a run-time dependent mapping array, VFCS generates appropriate code, which at run-time determines the set of local elements for each processor.

call graph of the program, in order to determine the distribution of dummy arguments. If interprocedural distribution analysis detects that a dummy array is associated with incompatible distributions in different incarnations of a procedure, the system solves this conflict by automatically generating different versions of the procedure (*procedure cloning*).

All the information concerning the distribution of an array is stored in *distribution descriptors* which are used also in subsequent compilation phases.

After data distribution analysis VFCS analyzes the code in order to classify each loop according to the parallelization strategy that will be applied to it. A loop is marked to be parallelized with the inspector-executor strategy (see below) if it has the `INDEPENDENT` directive and if it contains indirect array accesses (by means of indirection arrays or vector subscripts) or references arrays that may have an indirect distribution. Otherwise, a loop is marked to be parallelized with the overlap strategy.

Overlap Parallelization Strategy

Loops that are parallelized with the overlap strategy are transformed by VFCS in several steps. First, based on the information obtained during interprocedural distribution analysis the system automatically derives the work distribution according to the *owner computes paradigm* and inserts high-level communication statements to ensure that data is communicated between the processors as required. Work distribution is implemented by associating each assignment statement with a boolean guard called *mask* such that each processor only executes assignments to data owned by it.

The efficiency of this parallelization strategy heavily depends on data dependence analysis and *overlap analysis*. Overlap analysis attempts to determine at compile-time which non-local elements of a distributed array are used in a processor. For many regular computations, the precise pattern of non-local accesses can be computed and utilized both to determine the storage requirements for the array and to optimize communication.

Moreover, if data dependence analysis detects that a loop does not contain true data dependences (or if the user has specified an `INDEPENDENT` directive), the communication is optimized by extracting communication statements out of the loop and combining them. In a further step, masks are eliminated wherever possible; e.g. by adapting the bounds of loops. However, if a loop contains accesses to arrays via indirection arrays, the access patterns cannot be analyzed at compile time, and thus worst case assumptions have to be made which will lead to inefficient parallel code.

Inspector-Executor Parallelization Strategy

Loops that access arrays indirectly (via index arrays) cannot be handled with compile-time analysis techniques like the overlap concept with reasonably efficiency. Therefore, for the parallelization of such *irregular loops* VFCS relies on run-time analysis techniques based on the inspector-executor paradigm. Note, that the inspector-executor strategy is only applicable if the loop has an `INDEPENDENT` directive. The `INDEPENDENT` directive implies that any communication required for accessing data items owned by other processors can be performed before the loop. Similarly, non-local data items assigned by an iteration can all be communicated back to the processors that own them at the end of the loop.

With the inspector-executor strategy a loop is transformed into three parts: 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 irregular

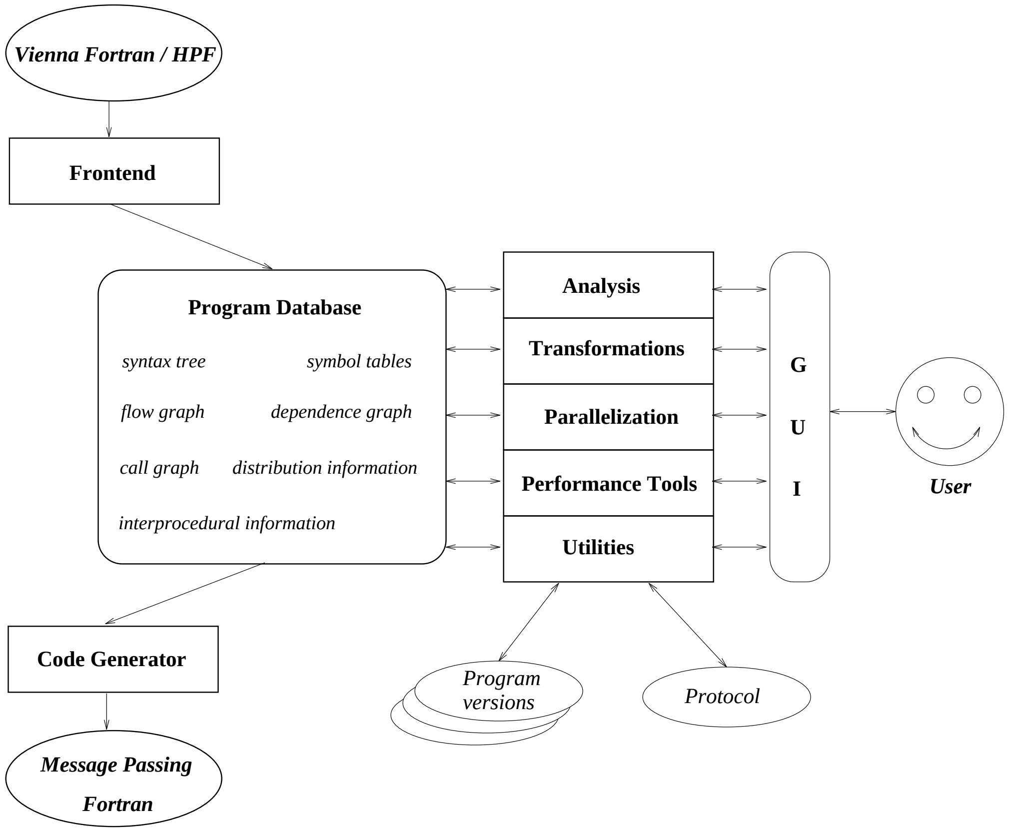


Figure 1: The Structure of VFCS

`INDEPENDENT` loop and generates a description of the communication pattern 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

Figure 1 shows the system structure of VFCS. The frontend component reads the input program and generates the internal representation which is kept in the *program database*. The program data base is accessed by all other system components. When VFCS is used in interactive mode the parallelization process may be directed by the user via the Motif-based graphical user interface.

The *analysis component* performs data flow analysis, data dependence analysis, interprocedural analysis and distribution analysis. The *transformation component* provides a catalog of normalization transformations and of standard transformations. The *parallelization component* provides various program transformations needed for the parallelization and vectorization of loops, and performs work distribution and communication generation. The *code generator* generates Fortran message passing code for various parallel target architectures. The integrated *Performance Tools* are a sequential profiler [12], which can be used to determine the performance critical sections of a program, a parameter-based performance prediction tool [11] that assists the user in fine-tuning the parallel program, and a tool for instrumenting the parallel program for performance analysis.

3 HPF and HPF+ Language Features

In this section we describe the HPF and HPF+ language features that are currently supported by VFCS. For a detailed specification of HPF+ please refer to the initial language definition [7].

3.1 Processor Arrays

HPF+ extends the handling of processor arrays in such a way that it is possible to establish a relationship between processor arrays, define different views of a processor array, and to specify and name subgroups of processor arrays.

In the current version of VFCS the 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(8)
!HPF+ PROCESSORS Q(4,2) = 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 [7], page 5), it is as if* `RESHAPE(R, SHAPE(Q))` *had been specified.*

VFCS currently requires that the size of a processor array is known at compile time.

3.2 Data Distribution

The following HPF+ distribution mechanisms are currently provided by VFCS:

- *Regular block distributions*
  A regular block distribution, `BLOCK`, partitions an array dimension to a processor dimension in evenly sized portions.

- *General block distributions*
  A general block distribution, `GEN_BLOCK`, partitiones 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.

- *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. Within the HPF+ project special emphasis has been paid on the implementation of `INDIRECT` distributions.

- *Replication*
  To indicate that an array dimension should be replicated, i.e. allocated on each processor, the "*" may be used. As in HPF, the number of distributed dimensions must always be equal to the rank of the corresponding processor array (section). Replication of data over processor dimensions and permutation of processor dimensions can only be achieved by means of alignment. This is also supported by VFCS.

- *Distribution to processor subsets*
  An array may be distributed to a regular processor array section. This relatively simple extension of the distribution mechanism allows the specification of mappings which otherwise could only be expressed by means of complex alignment specifications. Distribution to processor subsets has also been included in the HPF-2 Approved Extensions.

- *Multidimensional Distributions*
  Arbitrary dimensions of multidimensional arrays may be distributed using any combination of block and general block distribution and replication in the full generality as defined by HPF-2 and the HPF-2 Approved Extensions. If a multidimensional array is distributed indirectly, only a single array dimension, either the first or last, may be distributed.

- *Inheritance of Distributions*
  Dummy arguments of procedures always inherit their distribution from the actual arguments. Local arrays of procedures may be distributed in exactly the same way as arrays of the main program unit.

- *Distribution by alignment*
  The distribution of an array may be specified implicitly by aligning it with another (already distributed) array. VFCS supports most of the HPF alignment features, including perfect alignment, alignment offsets and strided alignment, permutation of axes, and collapsing and replication of dimensions.

Whereas `BLOCK` and `GEN_BLOCK` distributions currently must be defined statically (i.e. the arguments of these distribution functions must be defined at compile-time) `INDIRECT` distributions may be defined at run-time. Example 3.2 shows the data distribution features currently available in VFCS. Example 3.3 shows the alignment mechanisms provided by VFCS. The syntax of both the `DISTRIBUTE` and the `ALIGN` directive as implemented in VFCS is specified in the Appendix.

Example 3.2: HPF+ Distributions provided by VFCS

```
      INTEGER A2(100,100), A3(100,100,100)

!HPF$ PROCESSORS R(4), R2(2,2) = RESHAPE(R)


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


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


!     distribution to regular processor array sections
!
!HPF$ DISTRIBUTE(GEN_BLOCK(BS),BLOCK) ONTO R2(2:1:-1,:) :: A2


!     indirect distributions
!
      INTEGER MAP(100)
!HPF$ DISTRIBUTE(INDIRECT(MAP),*) :: A2
```

Example 3.3: Alignments provided by VFCS

```
!HPF$ ALIGN WITH A :: B             ! perfect alignment

!HPF$ ALIGN C(I,*) WITH A(I)        ! collapsing second axis of C

!HPF$ ALIGN D(I) WITH C(I,*)        ! replicating over second axis of C

!HPF$ ALIGN E(I,J) WITH F(J,I)      ! transposition of axes

!HPF$ ALIGN G(I) WITH H(2*I-1)      ! alignment strides and offsets
```

3.3 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 statments with the operators `+,-,*,` and `/` and the reduction functions `MIN` and `MAX` are implemented.

All these language features are supported by the VFCS with the same syntax and semantics as defined in HPF-2. Example 3.4 shows an `INDEPENDENT` directive with a `NEW` clause and a `REDUCTION` clause.

Example 3.4: `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` clause and the `DISTRIBUTE` clause may be used in an `INDEPENDENT` directive. With the `ON` clause the user may specify that a loop iteration should be executed on a particular processor or on a processor that owns a particular array element. By means of the `DISTRIBUTE` clause the loop iteration space may be distributed using the same distribution mechanisms as provided for arrays, i.e. loop iterations may be distributed using the `BLOCK`, `GEN_BLOCK`, `CYCLIC` and `INDIRECT` distribution function. All these work distribution mechanisms have been implemented in VFCS. Example 3.5 shows the use of the `ON` clauses and the `DISTRIBUTE` clause.

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

Example 3.5: `ON Clause and DISTRIBUTE Clause`

```
!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

!HPF$ INDEPENDENT, DISTRIBUTE(CYCLIC) ONTO R
      DO K=1,N
         ...
      END DO
```

In this example, iteration I of the first loop will be executed on the processor(s) 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))`. *The* distribute clause *of the last loop specifies that the iterations are mapped in a round-robin fashion to the processors.*

3.4 Pure and Purest Procedures

HPF+ introduces the `PUREST` directive which 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 kernel which calls from an `INDEPENDENT` loop a procedure that exclusively operates on *private* data objects. This is shown in Example 3.6.

3.5 Fortran 90 Features

A number of Fortran 90 language elements have been implemented within VFCS which originally was developed based on FORTRAN 77. Furthermore, a full Fortran 90/HPF frontend has been preliminarily integrated with VFCS and may be used optionally. Whereas this Frontend supports full Fortran 90, only selected features are currently handled by the core components of VFCS.

Currently VFCS supports the following Fortran 90 features:

- *Array Assignment Statements*

Example 3.6: `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`*.*

- *Array Sections*
  Array sections may be used in the context of array assignment statements and for referencing processor array sections in the `DISTRIBUTE` directive. Array sections must not be actual arguments of a procedure call.

- *Array Constants*
  Array Constants may be declared within the `PARAMETER` statement using a restricted form of array constructors, where each *ac-value* must be an integer constant.

- *Block DO Construct*
  The block `DO` construct may be used without the *named DO construct.*

- Fortran 90 free source form.

- Fortran 90 `INCLUDE` statement.

- Fortran 90 style of type declaration statements.

4 Compilation

4.1 Overview

In this section we describe the compilation techniques implemented in VFCS. In particular we given an overview of the overlap strategy, which is used to parallelize regular computations, and then describe in more technical detail the implementation of the inspector-executor strategy which is applied to irregular computations like those considered within the HPF+ project.

4.2 Processing of I/O Statements

In the first phase the sequential source program is split into a *host part* and a *node part*. Depending on the execution model used (see Section 5.4.2) the host part will, after compilation, be executed on the host computer or a specially designated node of the target system, referred to as *host node*. It performs global management tasks, such as requesting resources, loading code, and terminating program execution; it also handles all input/output. Hence the I/O and termination statements of the original program, and all control statements which these depend on, will remain in the resulting host part.

The node part which is still a sequential Fortran program, contains the actual computation; input/output is represented by communication with the host node. All I/O statements are removed from the node part: reads are converted to the original read, a SEND statement in the host and a RECEIVE in the node part; writes require the original write statement, a RECEIVE statement in the host and a SEND in the node part.

Program control flow is modeled in both codes: hence the node part may have to communicate with the host part to provide values of loop bounds, the result of conditionals, or values required in I/O operations. Error or end-of-file specifications in I/O statements also influence control flow, and require communication of values from the host to the node part. Optimizations are necessary to ensure that only those values are transmitted which are actually required by the recipient to perform its tasks in the correct order.

The host and node parts resulting from this phase are loosely coupled; all synchronization necessary between them at run time will be achieved through message passing.

4.3 Distribution Analysis

With the current version of VFCS, a dummy argument always inherits the distribution of the corresponding actual argument. As a consequence, a dummy array may be associated with different distributions in different incarnations of a procedure. *Distribution analysis* determines whether a dummy argument may be associated with different distributions at run-time, and performs transformations which simplify the procedure interface, if necessary.

Classification of Dummy Arrays

For each procedure, a set of *call vectors* is determined. Each call vector is associated with a single incarnation; it is defined as the vector of actual argument distributions that are associated with that particular incarnation. Dummy arrays are classified according to their distribution as

- *non-distributed* (replicated) arrays
- *standard distributed arrays*
  which have a standard distribution in all incarnations, and
- *non-standard distributed arrays*
  which may have non-standard distributions in some incarnations.

A *standard distribution* is a distribution that is constructed using any combination of *block* or *general block* distribution and replication. To determine this property for a particular dummy array, all calls to the procedure are examined. For a given incarnation, if the actual array is a standard distributed array and the dummy array is a *subarray* of the actual array, then the dummy array will also have a standard distribution in that incarnation. A dummy array is a *subarray* of the actual array if its rank is less than or equal the rank of the actual array, and the index ranges in the common dimensions are identical.

A *non-standard distribution* is an `INDIRECT` distribution or a distribution of a dummy array that is not a *subarray* of the corresponding actual array.

Arrays are further classified such that all arrays with identical shape and distribution are contained in the same *distribution class*. Only one distribution descriptor is generated for all arrays contained in the same distribution class.

4.4 Transformation of Procedure Interfaces

Parameterization of Procedures

If a dummy array is associated in different incarnations with different actual arrays which are not contained in the same distribution class (i.e. they have different standard distributions), then the procedure is parameterized by passing in addition to the array it's distribution class. Note that a parameterization of a procedure with a distribution class is not visible to the user before target code has been generated.

Cloning

If a dummy argument can be bound in different call vectors to a distributed as well as a non-distributed actual argument, this conflict is solved by *cloning*: two different versions of the procedure are automatically generated. In order to improve efficiency of the target code, the system tries to separate procedure incarnations in which a dummy array will have standard distributions from incarnations in which the same array will have non-standard distributions. This is also done by cloning.

4.5 Overlap Strategy

The overlap strategy, which was initially developed in the context of the `SUPERB` system [14, 15], is a parallelization method that has been designed with special focus on regular, nearest neighbor computations (e.g. stencil operations). For these codes the compiler can usually determine the precise access patterns for distributed arrays and the required communication structure and thus very efficient parallel code can be generated. Parallelization of loops with the overlap strategy is done in two phases. In the *basic parallelization* phase initial work distribution and communication insertion is performed. The subsequent optimization phase optimizes communication and work distribution of loops.

4.5.1 Initial Work Distribution

Masking is a transformation that according to the *owner computes* paradigm ensures that each processor executes only assignments to those elements of a distributed array that are owned by it. This is implemented by associating each assignment statement (except those that appear inside an irregular `INDEPENDENT` loop) with a boolean guard. During interactive parallelization a statement mask is indicated by prefixing each assignment statement with `OWNED(ref)->`, with the meaning that the assignment is executed only on those processors on which `OWNED(ref)` evaluates to `.TRUE.`. In subsequent phases, the system tries to eliminate masks wherever possible, i.e. by adapting the bounds of a surrounding loop accordingly. Masks that cannot be eliminated are translated into logical `IF` statements during the code generation phase.

4.5.2 Initial Communication Generation

If a processor executes a statement that references distributed data, it may require access to non-local data. Thus, during the basic parallelization phase, for all references which may access non-local variables a high-level communication statement `EXSR` (EXchange Send Receive) is inserted which updates private copies of the variables if necessary. In the basic parallelization phase, VFCS generates for each right-hand-side reference to a distributed variable an `EXSR` statement. `EXSR` statements provide an abstraction of the actual communication. During communication optimization, `EXSR` statements are extended by an *overlap description* that determines the storage requirements for the non-local variables. In the code generation phase `EXSR` statements are replaced by corresponding calls to a message passing library.

4.5.3 Overlap Analysis

Overlap analysis is performed to determine which non-local elements of a distributed array[2] are used in a processor. For many regular computations, the precise pattern of non-local accesses can be computed; this information can then be used both to determine the storage requirements for the array and to optimize communication. During overlap analysis the compiler determines for each distributed array and each reference to it an overlap description which gives feedback on the locality of certain computations and on the amount of non-local data required. Thus, overlap descriptions are also useful as a measure for the quality of an array's distribution.

[2] Overlap analysis is not performed for arrays with an `INDIRECT` distribution.

For each distributed array the compiler determines those parts which are mapped to a particular processor. The portion of an array that is mapped to a particular processor is referred to as its *local segment*. The *overlap area* of an array on a particular processor p, is the smallest rectilinear contiguous area around the local segment of the array on p containing all non-local elements accessed by p during the program. Note that an array's overlap area is a conservative approximation of the non-local elements accessed during execution of the program. The system represents overlaps by means of *overlap descriptions*. An overlap description for an n-dimensional array is presented in the form

$$[dl_1/du_1, \dots, dl_n/du_n]$$

where dl_i and du_i denote the offsets with respect to the lower and upper bound of dimension i of the local segment. The system displays overlap descriptions by appending them to the corresponding `EXSR` statements. Overlap descriptions are used by the system to significantly improve the organization of communication; they facilitate memory allocation and the local addressing of arrays. Note also that during overlap analysis unnecessary `EXSR` statements are eliminated. The relevant analysis is described in detail in [13].

4.5.4 Optimization

Optimization of loops which have been transformed according to the overlap strategy is performed in two steps:

Optimization of Communication

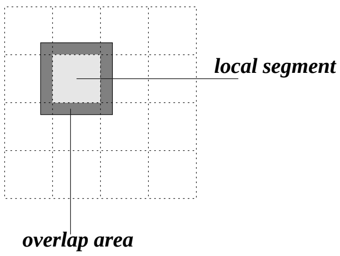
Optimization of communication extracts communication statements from regular `DO`-loops wherever possible and thus combines many single element messages to few larger ones. This optimization technique, also know as *communication vectorization*, minimizes the communication overhead by reducing both latency and startup time. It is applied if either dependence analysis reveals that the loop does not contain loop carried true dependencies or an `INDEPENDENT` directive has been specified by the user in the source code.

Optimization of Work Distribution

After initial work distribution each assignment statement is associated with a mask which ensures that only those processors which are owners of the right-hand-side variable execute the assignment. *Mask optimization* attempts to optimize the handling of masks in loops by strip-mining the loop across the processors. The iteration space is partitioned in such a way that each processor executes its own set of loop iterations which is achieved by enforcing the masks by an appropriate adaptation of the loop bounds in the internal representation. Adaptation of loop bounds as outlined above may not be applicable if a loop contains assignments to scalar variables. In such cases the loop should be transformed (i.e. by applying scalar forward substitution or induction variable substitution) prior to the optimizations steps.

In Figure 2 the transformation steps applied during parallelization with the overlap strategy are shown using the Jacobi kernel.

```
! original code
      REAL F(N,N), U(N,N), UHELP(N,N)
!HPF$ PROCESSORS R(4,4)
!HPF$ DISTRIBUTE(BLOCK,BLOCK) :: F, U, UHELP
      ...
      DO J=2,N-1
        DO I=2,N-1
           UHELP(I,J) = 0.25*(F(I,J)+U(I-1,J)+U(I+1,J)+U(I,J+1)+U(I,J-1))
        END DO
      END DO
---------------------------------------------------------------------------------
! code after initial masking and communication insertion
      DO J=2,N-1
        DO I=2,N-1
          EXSR U(I,J-1)
          EXSR U(I,J+1)
          EXSR U(I+1,J)
          EXSR U(I-1,J)
          EXSR F(I,J)
          OWNED(UHELP(I,J))-> UHELP(I,J) = 0.25*(F(I,J)+U(I-1,J)
     *                             +U(I+1,J)+U(I,J+1)+U(I,J-1))
        END DO
      END DO
---------------------------------------------------------------------------------
! code after overlap analysis
      DO J=2,N-1
        DO I=2,N-1
          EXSR U(I,J-1)  {[0/0,1/0]}
          EXSR U(I,J+1)  {[0/0,0/1]}
          EXSR U(I+1,J)  {[0/1,0/0]}
          EXSR U(I-1,J)  {[1/0,0/0]}
          OWNED(UHELP(I,J))-> UHELP(I,J) = 0.25*(F(I,J)+U(I-1,J)
     *                            +U(I+1,J)+U(I,J+1)+U(I,J-1))
        END DO
      END DO
```




```
---------------------------------------------------------------------------------
! after communication optimization and mask optimization
      EXSR U(:,:) {[0/0,1/0]}
      EXSR U(:,:) {[0/0,0/1]}
      EXSR U(:,:) {[0/1,0/0]}
      EXSR U(:,:) {[1/0,0/0]}
      mask-> DO J=2,N-1
        mask-> DO I=2,N-1
           UHELP(I,J) = 0.25*(F(I,J)+U(I-1,J)+U(I+1,J)+U(I,J+1)+U(I,J-1))
        END DO
      END DO
```

Figure 2: Overlap Strategy - This example shows the jacobi kernel obtained after basic parallelization, after overlap analysis, and after optimization. The overlap descriptions are appended to the corresponding `EXSR` statements. The overlap area and the local segment for array `U` on a particular processor is visualized.

4.6 Inspector-Executor Strategy

The compilation techniques applied for parallelizing irregular `INDEPENDENT` loops differ significantly from those applied to other parts of the program. Since in such loops arrays are usually accessed indirectly (via index arrays) compile–time analysis techniques like the overlap concept cannot be applied with reasonably efficiency. Thus, irregular `INDEPENDENT` loops are not modified during the basic parallelization phase and the optimization phases which were described in the previous sections.

Whereas with the overlap strategy the work distribution is derived automatically from the data distribution according to the owner computes paradigm, the distribution of loop iterations of irregular `INDEPENDENT` loops may be specified explicitly by the user by providing a work–distribution clause. Thus with this strategy a processor may also compute data items which it does not own. If the user does not specify the distribution of loop iterations, the system chooses a default work distribution. VFCS automatically parallelizes *irregular* `INDEPENDENT` loops according to the *inspector–executor* paradigm[19], which is described in more detail below.

In the context ot the `INDEPENDENT` directive VFCS supports the HPF-1 `NEW` clause and the HPF–2 `REDUCTION` clause. `NEW` specifies the private variables (local in each iteration). For scalar variables the `REDUCTION` clause allows to perform global operations across a set of iterations, and for distributed arrays to accumulate the values onto the array elements. In the current implementation of `INDEPENDENT` loops all scalars and non-distributed arrays defined within the loop must be included either in the `NEW` or `REDUCTION` clause.

4.6.1 Overview

The process of transforming an irregular `INDEPENDENT` loop combines compile time parallelization techniques with runtime analysis. Information from dataflow analysis, data dependence analysis, data distribution and overlap analysis is utilized. To implement runtime analysis and parallel execution, a special runtime library which has been developed based on the PARTI [9] primitives from ICASE NASA is used. The parallelization techniques utilize the fact that an `INDEPENDENT` directive has been specified and the execution of an iteration cannot change data items used by any other iteration. Thus, any communication required for accessing data items owned by other processors can be performed before the loop, and similarly, non–local data items assigned by an iteration can all be communicated back to the processors that own them at the end of the loop.

With the inspector-executor parallelization strategy, an independent `DO` loop loop is transformed into three main phases: the *work distributor*, the *inspector*, and the *executor* as depicted in Figure 3. Furthermore, for each class of distributed arrays referenced in the loop *constructor of data distribution descriptor* code for constructing a runtime distribution descriptor is generated.

4.6.2 Runtime Distribution Descriptors

The runtime distribution descriptor (RDD) for a distributed array A, includes all information about the size, shape, distribution type, and, in the case of indirectly distributed arrays, two additional data structures which contain the *local index set*, and the *translation table*.

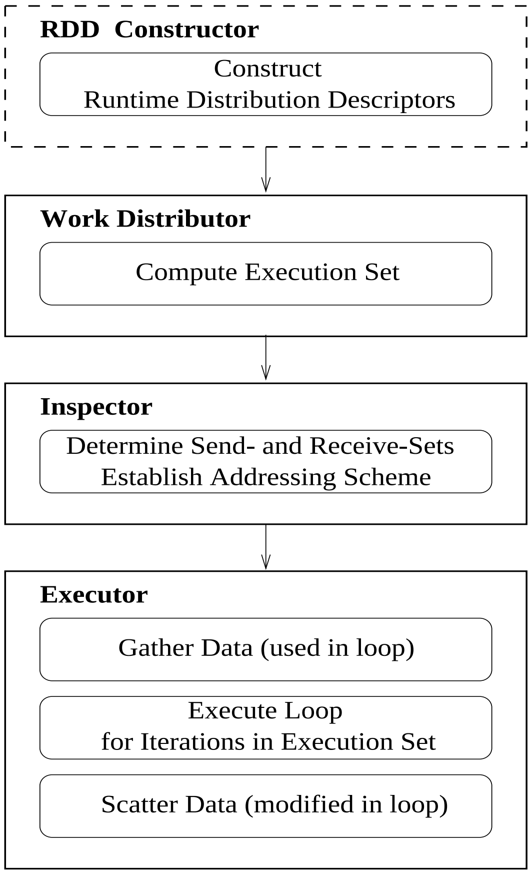


Figure 3: Phases Generated for a DO INDEPENDENT Loop

Definition 4.1: *Local Index Set* and *Local Segment*

The local index set of an array `A` *in each processor* `p`*, denoted by* $Local^A(p)$*, is an ordered set of indices designating those elements of the array* `A` *that have to be allocated on that particular processor* `p`*. An array element* `A(i)` *is said to be owned by processor* `p`*, or* `p` *is the home processor of* `A(i)`*, iff* $\mathtt{i} \in Local^A(p)$*.*

The set of elements of `A` *that are allocated in the local memory of a processor* `p` *is called the local segment of* `A`*.* ■

Definition 4.2: `Translation Table`

The translation table for an array `A`*, is a distributed data structure, denoted by* $Trat^A$*, which records the home processor and the local address in the home processor's memory for each array element* `A(i)`*.* ■

```
      INTEGER :: localA_size                        ! size of local index set
      INTEGER, DIMENSION(:), POINTER :: localA      ! local index set
      TYPE (TRA_TAB), POINTER :: ttA                ! translation table
      TYPE (DIST_DSC) :: ddA                        ! data distribution descriptor
      ...
C   Constructing local index set for A using the values of MAP
      CALL Build_LIS(ddMAP, MAP, localA, localA_size)

C   Constructing translation table for A
      ttA = Build_Trat(localA, localA_size)

C   Constructing runtime distribution descriptor for A
      ddA%local = localA
      ddA%local_size = localA_size
      ddA%trat = ttA
C     ... Initialization of other fields of ddA ...
```

Figure 4: RDD constructor for array A

Figure 4 illustrates[3] the code for constructing the runtime distribution descriptor denoted dd^A, for an array A which is indirectly distributed according to the mapping array MAP. It is supposed that the distribution descriptor of MAP, dd^{MAP}, has been already initialized.

4.6.3 Buffer Management

A processor may access only its private address space. So, if an array element is referenced which is not local, the processor must retrieve a copy of that element from the home processor by using some communication mechanism. The processor stores this copy in its local memory called the *overlap area* (when applying the overlap strategy), or the *buffer space* (when applying the inspector–executor strategy). The local memory allocated for an array `A` on processor p consists of the local segment of `A` and the corresponding communication area.

Consider three arrays `A, B` and `C` with the following distribution:

```
        INTEGER, PARAMETER :: MAP(16) = (/1,2,2,3,3,3,4,4,4,4,3,3,3,2,2,1/)
        REAL, DIMENSION(16) :: A, B, C
!HPF$   PROCESSORS :: PROC(4)
!HPF$   DISTRIBUTE(INDIRECT(MAP)) ONTO PROC :: A
!HPF$   DISTRIBUTE(BLOCK) ONTO PROC :: B, C
```

Suppose the array `A` occurs in irregular code, the array `B` occurs only in regular code and array `C` in

[3] For sake of readability we use Fortran 90 derived types for describing the data structures that are contained in a runtime distribution descriptor. In the current implementation these descriptors are implemented as a set of Fortran 77 variables.

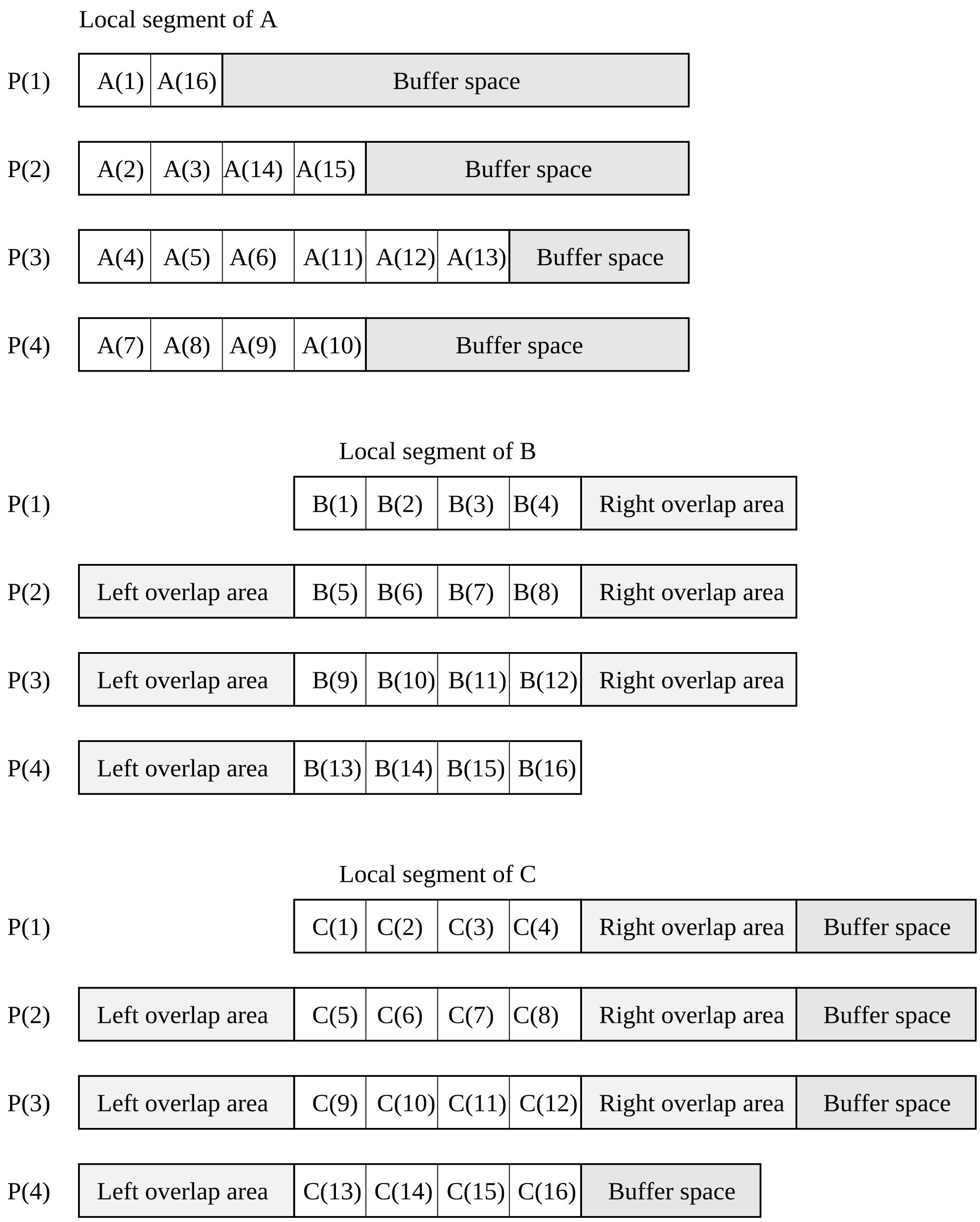


Figure 5: Organization of Local Segments a Communication areas

both regular and irregular code. Then the structure of the allocated local memories for arrays `A, B` and `C` is organized as illustrated in Figure 5.

The parameters in the formulas have the following meaning:

NP	...total number of processors
p	...index of an individual processor, $0 \le p \le (NP-1)$
N	...size of the global iteration set, $N = U - L + 1$
$[L:U]$	...a contiguous range of integers $\{ i \mid L \le i \le U \}$

!HPF$ INDEPENDENT, DISTRIBUTE(BLOCK)
    DO i = L,U

Iterations are divided into contiguous blocks, whose sizes differ by at most 1.
Let $q = \lfloor \frac{N}{NP} \rfloor$, $r = MOD(N, NP)$.
If $N < NP$ then
    $exec(p) = \{p + L\}$, for $p < N$
    $exec(p) = \phi$, for $p \geq N$
else
    $exec(p) = [q * p + L : q * (p+1) + L]$, for $p < r$
    $exec(p) = [q * p + L : q * (p+1) - 1 + L]$, for $p \geq r$

!HPF$ INDEPENDENT, ON HOME(A(f(i)))
    DO i = L,U

Iteration i is executed on the processor which owns the array element A(f(i)).
If f is an invertible function then
    $exec(p) = f^{-1}((local^A(p)) \cap [L:U])$
where, local^A(p) denotes the set of local elements of the array A on processor p.

!HPF$ INDEPENDENT, DISTRIBUTE(CYCLIC)
    DO i = L,U

Iterations are distributed in a round robin fashion across the processors.
    $exec(p) = [p + L : U : NP]$

!HPF$ INDEPENDENT, ON PD1(g(i))
    DO i = L,U

Iteration i is executed on the processor $g(i)$. If g is an invertible function then
    $exec(p) = g^{-1} \cap [L:U]$

Figure 6: Strategies for Determination of the Execution Set

4.6.4 Work Distributor

On each processor `p`, the work distributor computes the *execution set exec(p)*, i.e. the set of loop iterations to be executed on processor `p`. Depending on the work distribution specification, the compiler selects different ways for the determination of *exec(p)* as shown in Figure 6.

The descriptor for the execution set `ESD` can be of the form as depicted in Figure 7. If the execution set can be expressed by a regular section it is represented by a triplet, else it is represented as one-dimensional integer array. The basic algorithm for implementing the work distributor which calculates the execution set according to the owner of an array element `A(f(i))` as shown in Figure 6

```
TYPE ESD                                        ! execution set descriptor
   LOGICAL :: isregular                         ! indicates whether execution set is regular
   INTEGER :: low, up, step                     ! regular section description
   INTEGER :: size                              ! size of the execution set
   INTEGER, DIMENSION(:), POINTER :: execp      ! execution set
END TYPE ESD
```

Figure 7: Execution Set Descriptor

is outlined in Figure 8. Is_Local is a Boolean function which yields the value `TRUE` iff the array element `A(f(i))` is owned by the executing processor.

```
TYPE(DIST_DSC) :: dd^A                          ! distribution descriptor of A
INTEGER :: exec_size                            ! size of the execution set
INTEGER, DIMENSION(:), ALLOCATABLE :: exec      ! execution set
...
ALLOCATE(exec(dd^A%local_size))
exec_size = 0
DO i= L, U
   IF (Is_Local(dd^A, f(i))) THEN
      exec_size = exec_size + 1
      exec(exec_size) = i
   END IF
END DO
```

Figure 8: Work Distributor

4.6.5 Inspector and Executor

The inspector performs runtime analysis of the `DO INDEPENDENT` loop; it generates a description of the communication necessary for the loop in the form of *schedules*. Schedules control the runtime procedures that move data among processors in the subsequent executor phase, and establish an addressing scheme between the global and local array accesses. Information needed to generate a schedule, to allocate a communication buffer, and for the global to local index conversion is obtained from the appropriate *global reference lists*, along with the knowledge of the array distributions.

The *global reference lists* are constructed from the subscript functions of the accessed arrays. The subscripts are collected by means of a do loop iterating over the computed execution set. Symbolic analysis ensures that only textually distinct subscripts are included in the list. In the case of a multi-level indirection, the array accesses are processed in multiple phases, starting with the innermost arrays. This order guarantees that non-local data references occuring in a subscript expression of an array are resolved in previous phases. Results of global to local index conversion are stored in the corresponding *local reference lists*. The local reference lists are constructed in such a way that the local index points either to the local segment or to the buffer.

The *communication schedule* contains the necessary information to identify the locations in distributed memory from which data is to be acquired.

The `executor` is the final phase in the execution of the transformed `DO INDEPENDENT` loop; it performs communication described by schedules (gather/scatter non–local array elements used/updated in the loop), and executes the loop body according to the execution set. The schedules control communication in such a way that execution of the loop using the local reference list accesses the correct data in local segments and buffers. The loop body is transformed such that every global subscript of a distributed array is replaced by the corresponding local reference.

4.6.6 Optimizations Techniques

Runtime analysis techniques as outlined above might introduce a high overhead. Therefore, `DO INDEPENDENT` loops which include only regularly distributed arrays with regular accesses are parallelized by applying the more efficient overlap strategy. The first step in the parallelization of `INDEPENDENT` loops according to the inspector–executor strategy is the construction of high–level data structures which serve as a framework for subsequent compile–time optimizations and for target code generation. The primary goal of the optimization phase is to identify situations that allow the elimination of redundant runtime preprocessing. The cost of the inspector phase can be amortized when the information generated by the inspector can be computed only once and then used repeatedly.

In a symbolic analysis phase arrays occurring in the loop are classified according to their shape, distribution and access patterns. Only one schedule is generated for all arrays contained in the same class.

4.6.7 Example

The inspector–executor parallelization technique will be illustrated on the example given in Fig. 9. It is derived from the unstructured mesh linear equation solver `GCCG` which is a part of the program package FIRE.[4] Loop L represents the kernel loop of this solver.

Array `D1` is indirectly distributed using the mapping array `MAP` which is read at runtime. The remaining arrays have a `BLOCK` distribution. The main loop L which is contained within an outer iteration cycle computes a new value for every cell by using the old value of this cell and of its six indirectly addressed neighbor cells. The specified work distribution, `ON HOME(D2(i))`, ensures that communication is caused only by references to array `D1`. The references to the other arrays (`BP, BS, BW, BL, BN, BE`, and `BH`) are local because these arrays have the same distribution and access patterns as array `D2`.

The runtime distribution information generated for the array D1 is shown in Figure 10. The inspector and executor generated for the loop is introduced in Figures 11 and 12, respectively.

[4]FIRE is a registered trade-mark of AVL LIST Gmbh.

```
      PARAMETER :: NP = ..., NNINTC = ..., NNCELL = ...
      DOUBLE PRECISION, DIMENSION(NNINTC) :: D2, BP, BS, BW, BL, BN, BE, BH
      DOUBLE PRECISION, DIMENSION(NNCELL) :: D1
      INTEGER, DIMENSION(NNINTC,6) :: LCC
      INTEGER, DIMENSION(NNCELL) :: MAP

!HPF$ PROCESSORS PROC(NP)
!HPF$ ALIGN WITH D2 :: BP, BE, BW, BN, BS, BH, BL
!HPF$ ALIGN LCC(I,*) WITH D2(I)
!HPF$ DISTRIBUTE(BLOCK) ONTO PROC :: D2
!HPF$ DISTRIBUTE(BLOCK) ONTO PROC :: MAP
!HPF$ DISTRIBUTE(INDIRECT(MAP)) ONTO PROC :: D1
        ...
!HPF$ INDEPENDENT, ON HOME(D2(i))
L:    DO i = 1, NNINTC
          D2(i) = BP(i) * D1(i) -                   &
                  BS(i) * D1(LCC(i,1)) -            &
                  BW(i) * D1(LCC(i,4)) -            &
                  BL(i) * D1(LCC(i,5)) -            &
                  BN(i) * D1(LCC(i,3)) -            &
                  BE(i) * D1(LCC(i,2)) -            &
                  BH(i) * D1(LCC(i,6))
        END DO
```

Figure 9: Kernel Loop of the GCCG Solver

In the first step the execution set according to the `ON HOME` clause is generated and stored in an integer array *exec*; *exec_size* saves its cardinality. The global reference lists, $globref_i$, $i = 1, \ldots, 3$, are computed from the subscript functions and from the execution set (see lines *G1–G16*). The global reference list, $globref_i$, its size and the distribution descriptor of the referenced array are passed to the procedure Localize (line *I3*) to determine the appropriate schedule $sched_i$, the size $nonloc_i$ of the communication buffer to be allocated, and the local reference list $locref_i$ which contains results of global to local index conversion. The array declarations in the message passing code allocate memory for the local segments holding the local data and the communication buffers storing copies of non–local data. The procedure Ind_Conv (lines *I1, I2*) performs the global to local index conversion. In the executor phase, only non–local elements of the array `D1` must be gathered by the runtime procedure dGather (line *E1*); the work and data distribution specifications guarantee that all other array accesses are local.

```
      TYPE(DIST_DSC) :: dd0       ; data distribution descriptor for MAP
      TYPE(DIST_DSC) :: dd1       ; data distribution descriptor for LCC
      TYPE(DIST_DSC) :: dd2       ; data distribution descriptor for D2, BP,... BH
      TYPE(DIST_DSC) :: dd3       ; data distribution descriptor for D1
      INTEGER, POINTER      :: lis3(NNINTC)   ; local index set for D1
      INTEGER               :: lis_size3      ; size of lis3
      TYPE(TRA_TAB), POINTER :: tt3           ; translation table pointer for D1

C  Constructing local index set and translation table for descriptor dd3
      CALL Build_LIS(dd0, MAP, lis3, lis_size3)
      tt3 = Build_Trat(lis3, lis_size3)
      CALL Update_RDD(dd3, lis3, tt3)
```

Figure 10: RDD for the GCCG Kernel Loop

4.7 Runtime Support for Irregular Codes

Much of the complexity of the inspector executor-parallelization strategy is contained in the runtime library for irregular computations which has been developed based on an enhanced version of the PARTI ([9]) and CHAOS ([10]) runtime routines.

The PARTI routines were originally developed as a runtime support for a class of irregular problems which are characterized by a sequence of concurrent computational phases whose data access patterns and computational cost cannot be determined at compile-time. They were designed to ease the implementation of irregular problems on distributed memory parallel architectures by relieving the user from having to deal with many low–level machine specific issues. The PARTI routines can be used in tasks such as global–to–local index mappings, communication schedule generation, and schedule based communication. This functionality can be used directly to generate inspector–executor code.

The CHAOS runtime library is a superset of the PARTI designed to partitioning data automatically. It provides efficient support for parallelization of adaptive irregular problems where indirection arrays are modified during the course of the computation.

4.7.1 Overview of Runtime Procedures

The inspector–executor code generated by VFCS is built using the following PARTI routines:

- `build_translation_table()` – constructs the distributed data structure (see Definition 2) which for each array element lists the home processor and the local address. Each processor passes to `build_translation_table` the local index set (see Definition 1) and its size. In the current implementation the translation tables are block distributed.
- `localize()` – translates the global indices of an irregularly distributed array into local ones and generates the schedule which is used to gather and scatter data. Each processor passes to

```
C  INSPECTOR phase

C  Constructing the execution set
S1:     CALL Build_ES_owner(dd2, 1, NNINTC, exec, exec_size)

C  Constructing global reference list for the LCC
G1:     n1 = 1
G2:     DO k = 1, exec_size
G3:        globref1(n1) = exec(k); n1 = n1 + 1
G4:     END DO

C  Index Conversion for LCC having the data distribution descriptor dd1
I1:     CALL Ind_Conv(dd1, n1-1, globref1, locref1)

C  Constructing global reference lists for D1, D2, BP, BS, BW, BL, BN, BE, BH
G5:     n1 = 1; n2 = 1; n3 = 1
G6:     DO k = 1, exec_size
G7:        globref2(n2)   = exec(k); n2 = n2 + 1
G8:        globref3(n3)   = exec(k)
G9:        globref3(n3+1) = LCC(locref1(n1),1)
G10:       globref3(n3+2) = LCC(locref1(n1),4)
G11:       globref3(n3+3) = LCC(locref1(n1),5)
G12:       globref3(n3+4) = LCC(locref1(n1),3)
G13:       globref3(n3+5) = LCC(locref1(n1),2)
G14:       globref3(n3+6) = LCC(locref1(n1),6)
G15:       n3 = n3 + 7; n1 = n1 + 1
G16:    END DO

C  Index Conversion for D2, BP, BS, BW, BL, BN, BE, BH
C    having the common data distribution descriptor dd2
I2:     CALL Ind_Conv(dd2, n2-1, globref2, locref2)

C  Computing schedule and local reference list for D1
I3:     CALL Localize(dd3, n3-1, globref3, locref3, nonloc3, sched3)
```

Figure 11: Inspector for the GCCG Kernel Loop

```
C   EXECUTOR phase

C   Gather non-local elements of D1
        nbuf = ...           ! begin of buffer
E1:     CALL dGather(D(nbuf), D(1), sched3)

C   Transformed loop
        n2 = 1; n3 = 1
        DO k=1, exec_size
           D2(locref2(n2)) =  BP(locref2(n2)) * D1(locref3(n3)) -     &
                              BS(locref2(n2)) * D1(locref3(n3+1)) -   &
                              BW(locref2(n2)) * D1(locref3(n3+2)) -   &
                              BL(locref2(n2)) * D1(locref3(n3+3)) -   &
                              BN(locref2(n2)) * D1(locref3(n3+4)) -   &
                              BE(locref2(n2)) * D1(locref3(n3+5)) -   &
                              BH(locref2(n2)) * D1(locref3(n3+6))
           n2 = n2 + 1; n3 = n3 + 7
        END DO
```

Figure 12: Executor for the GCCG Kernel Loop

`localize` the relevant translation table, a list of global references, the number of global references and information about the shape of the array. The procedure returns the corresponding list of local references, a schedule and the number of non–local data elements.

- `reglocalize()` – the functionality of this procedure is similar to that of `localize` - it translates the global indices of a regularly distributed array into local indices. Instead of passing a translation table as argument to the procedure a regular distribution type must be passed.

- `PREFIXgather()`, `PREFIXscatter()` – data exchangers procedures; `PREFIX` can be i (integer), f (real), d (double precision) or c (character). Gather procedures place copies of data obtained from other processors in the buffer space, scatter procedures write the copies of data placed in the buffer back to the home processor. Each processor passes to the procedure the schedule produced by `reglocalize` or `localize` together with the addresses of the local segment and buffer.

- `PREFIXscatter_FUNC()` – is used for implementing *accumulation* operations specified with a `REDUCTION` clause.

- `PREFIXmultigather()`, `PREFIXmultiscatter()` – these procedures are an extension of the PREFIXgather/PREFIXscatter procedures, such that they allow to specify multiple schedules to be used, and they allow multi–column or multi–row gathers/scatters if the columns (rows) are distributed in the same way.

- `PREFIXmultiscatter_FUNC()` – these procedures are an extension of the PREFIXscatter_FUNC() procedures, such that they allow to specify multiple schedules to be used, and they allow multi–column or multi–row scatters if the columns (rows) are distributed in the same way.

4.7.2 Extensions for HPF+

The original `reglocalize` procedure supported only Vienna Fortran `BLOCK` and `CYCLIC(1)` distributions. This procedure was extended for processing of HPF `BLOCK`, `GEN_BLOCK`, and `CYCLIC(n)` distributions. For the handling of HPF `INDIRECT` distributions the procedure `Build_LIS()` was implemented. It constructs for an irregularly distributed array the local index set which is the basis for the generation of the appropriate translation table. Each processor passes to `Build_LIS` the mapping array, either the whole array, if it is replicated, or its local segment together with its distribution descriptor. Furthermore, a number of routines providing a suitable interface between the target Fortran program and the runtime routines were implemented.

The original PARTI library is based on the NX library which has no support for advanced concepts like communicators and global communications that are provided in MPI [21]. Therefore, and because MPI has been chosen by the HPF+ Consortium as the message passing platform, it was necessary to port PARTI to MPI. This has a number of important advantages:

- Complex global communications can be implemented in MPI more easily.
- MPI communicators provide a means to perform collective communications only among those processors that own a part of data. As an example, distributing data onto processor sections can be efficiently implemented using communicators.
- As MPI is a hardware independent library and an implementation of it exist on every distributed memory platform, the codes using MPI are considered to be highly portable.
- MPI also provides an instrumented version of communication routines with the same interface. This removes an extra effort to manually instrument the runtime system.

In the original PARTI library the global communications have been implemented by coloring the processors and exchanging null byte messages (as a means for synchronization) between every two processors that have the same color before actual messages are exchanged. Such complex communication schemes - otherwise proved to be correct - always represent potential sources of deadlock. Hence, this mechanism was replaced with global communication routines such as `MPI_ALLTOALL()` whenever sufficient information was available about the send/receive sets. Examples of such cases in PARTI routines are: `execute_gather()`, `execute_scatter()`, etc.

In those situations where `MPI_ALLTOALL()` could not be used (because either the communication had a point-to-point nature or it was a global communication but the size of messages to be received was not known) the point-to-point routines of the NX library were replaced with their MPI equivalents.

5 Using VFCS

In this section we describe how to use VFCS for the parallelization of HPF programs either in automatic mode, as a command-line compiler, or in interactive mode where the user can control the parallelization process via a Motif-based graphical user interface.

5.1 Automatic Parallelization with VFCS

VFCS, when invoked in automatic mode, is executed under the control of a driver file. Default driver files are provided for different parallel target architectures. A set of directives, the *driver directives*, enables the user to specify the sequence of program transformations applied during the parallelization process. Thus, the user is able to customize the system to meet specific properties of the application to parallelize. Driver files may be created manually using the driver directives as described in the VFCS User's Guide [6]. Furthermore, driver files may be generated automatically by protocoling an interactive parallelization session.

VFCS Command-Line Syntax

The VFCS command-line syntax is

vfcs [*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.

If *filename* is omitted, VFCS is executed in interactive mode

If *filename* is present but neither the -*a* nor the -*b* option is specified, VFCS is executed in automatic mode using the default driver file.

Options is zero or more of:

-*a drvfile* execute VFCS in automatic mode;
drvfile indicates the driver file to use;

-*b drvfile* same as -*a* option; see above

-*p project* use *project* as prefix for the names of the generated message passing source file(s)

-*v* set verbose output for automatic mode

VFCS also accepts the standard Toolkit options (see X(1)).

5.2 Interactive Parallelization with VFCS

This section describes the use of VFCS in interactive mode and discusses the major steps of transforming a sequential program into a parallel message-passing program. When used in interactive mode, VFCS provides a set of analysis services and a transformation catalog enabling the user to guide the system through the parallelization process by means of a hierarchically structured menu system based on the X11-MOTIF user interface.

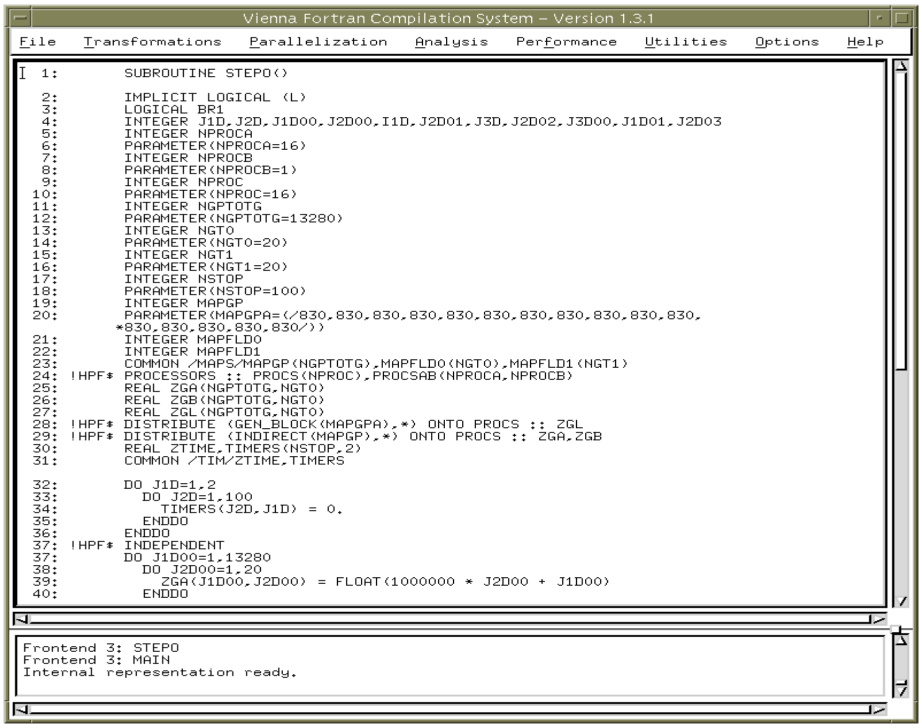
In this report we do not discuss all features provided by VFCS but only those that must be used for parallelizing an HPF program. For a detailed description of all available menu items refer to the VFCS User's Guide [6].

VFCS is invoked in interactive mode by omitting the source file(s) at the command-line, by typing:

vfcs

5.2.1 Menu Structure

After VFCS has been started, the VFCS main window as shown in Figure 13 is displayed. The VFCS main menu is presented in the second line of this window. Below, in the text sub-window, the source code of the actual program unit is displayed. The message sub-window is used by VFCS for displaying status messages. The available menu items of VFCS are shown in Figure 14.



← Text Window

← Status Window

Figure 13: VFCS Main Window

File	Transformations	Parallelization	Analysis	Performance	Utilities	Options
Load Source Files	Induction Variable	Data Distribution	Data Flow	Weight Finder	Units	Code Generation
Save Target Files	Scalar Forward Subst.	Basic Parallelization	Data Dependence	P^3T	Goto Line	Comments On
Version Management	Mark Reduction	Modify Work Distribution	Call Graph	VFPMS	Search	Comments Off
Select Project Name	Inline Expansion	Communication Optimization	Procedure Calls		History	FORTRAN 77
Exit	Loop Distribution	Mask Optimization	Common Blocks		Protocol On	F90 fixed
	Loop Interchange	Code Generation	Data Distribution		Protocol Off	F90 free
	Loop Unrolling		Overlap			
	Unroll Jam					
	Mask Propagation					
	Expr. Simplification					
	Dead Code Elimination					
	Create Vector Code					
	Destroy Vector Code					

Figure 14: VFCS - Menu Structure

5.2.2 Main Steps in Parallelization

In the following we discuss the major compilation steps by which VFCS transforms a sequential program into a data parallel message passing program during an interactive parallelization session. For a discussion of the relevant compilation and analysis techniques refer to Section 4.

The following steps have to be performed in order to parallelize a Vienna Fortran or HPF program. All the corresponding menu entries are contained in the *Parallelization* menu. If the source program does not contain data distributions, these have to be added before step 2 using the interactive data distribution facility which may be invoked by choosing the *Data Distribution* menu item.

Step 1: Load Source Files

Reads in the source files of a program and performs scanning, parsing, program normalization, constant propagation, dependence analysis, semantic analysis and generates an internal representation of the program. The user may choose between two different frontends, a Fortran 77/Vienna Fortran/HPF frontend and a Fortran 90/HPF frontend, by selecting the appropriate entry form the *Options* menu.

Step 2: Basic Parallelization

Performs splitting (processing of I/O statements), interprocedural distribution analysis, transformation of procedure interfaces, and initial adaptation of regular loops according to the overlap parallelization strategy.

Step 3: Optimization

Communication optimization eliminates redundant communication, hoists communication statements out of loops wherever possible, and combines small messages into larger ones, based on the results of overlap analysis and dependence analysis.
Mask optimization adapts loop bounds according to the distribution of arrays defined in loops.

Step 4: Code Generation

In this phase irregular loops are transformed according to the inspector-executor strategy. Finally, target architecture specific Fortran message passing code is generated. This comprises adaptation of declarations and references to distributed arrays according to the data distribution, transformation of high level communication statements (EXSR) into calls to target architecture specific communication procedures, and elimination of *masks* by adapting loop bounds or generating conditional statements.

5.2.3 Transformations

VFCS provides an integrated program transformation tool which enables the user to adapt the program in such a way that parallelization transformations applied subsequently yield better results. These *standard transformations* can be selected manually from the *Transformations* menu and usually should be applied before step 2 (*basic parallelization*).

The following standard transformations are available:

- constant propagation
- scalar forward substitution
- induction variable substitution
- inline expansion
- dead code elimination
- loop transformations

The loop transformations include loop distribution, loop fusion, loop interchange, loop unrolling, unroll and jam, strip mining, and reduction recognition.

5.2.4 Utilities

Version Management

VFCS provides support to manage different versions of a parallelized program. By using the version management of VFCS the user may save the current state of the transformed program together with all the analysis information. The parallelization process can be restarted at that position by restoring the version. This is important since for most program transformations it is usually not feasible to provide an *undo facility.* Note that it is also possible to load a specific version of a program during a subsequent session, since program versions are saved until explicitly deleted by the user.

5.3 Vienna Fortran Performance Measurement System

VFPMS is an interactive environment for performing semi-automatic measurements of Vienna Fortran or HPF programs. It is implemented as a fully integrated module of VFCS. The user may specify measurements of interesting program components such as subroutines, loops, ranges, and statements during an interactive VFCS session. It is possible to measure execution time as well as communication time[5] for the selected program components. VFPMS automatically computes the required measuring points and generates an instrumented message passing FORTRAN node program as well as a detailed measuring description file.

[5]VFPMS supports only the measurement of regular communication statements (EXSR statements).

The specification of measurements has to be performed after parallelization and optimization of the program, but before *Target Code Generation.* Measuring instrumentation is actually performed after code generation in VFCS. Before code generation information on measurement progress can be visualized by using the *component browser*. VFPMS includes a context sensitive help system providing information in every situation within an user session.

Running the instrumented program on a parallel architecture produces a measurement file which can be used together with the measuring description for post execution performance analysis and monitoring.

5.4 Target Compilation and Execution of Parallelized Program

VFCS generates code for various parallel target architectures and supports several different communication libraries. These may be selected by the user via the menu entry *Code Generation* from the *Options* menu. The following options are currently available:

Target architectures:

- Intel iPSC/860
- Intel Paragon
- Meiko CS-2

Communication libraries:

- Intel NX
- Parmacs 6.0
- MPI

Execution models:

- Host-Node Model
- Node-Only Model

5.4.1 Output Files

VFCS creates several files in the working directory. Most output filenames may be specified by the user by means of command-line options. If the user does not specify output filenames, the system uses by default the following filenames for the generated parallel program.

host.f	contains the host program
node.f	contains the SPMD node program
*init.**	contains setup information for the parallel program. The size and contents of these files depend on the structure of the program. They are read by the host program upon program start.
Makefile, vfbuild	These files manage all necessary actions to generate the runtime environment for executing the program on the target system

If the *-p project* option has been specified at the command-line or a project name has been chosen interactively, all above filenames, except Makefile and buildfile, are prefixed by *project_*. The name of the makefile is changed to Make.*project*, and the name of the buildfile to vfbuild.*project*.

5.4.2 Target Compilation

VFCS generates a buildfile and a makefile, customized for the desired target system. They are used to set up the required VFCS runtime environment variables and to compile the output of VFCS on the target architecture. The customization is done by replacing a set of metavariables in architecture and system software independent buildfile and makefile templates. Running the buildfile manages all necessary actions to generate a runtime environment on the target system. It controls the establishment of the VFCS runtime environment, the target compilation, and the linking steps. In the following we assume that our project name has been set to *project*.

To generate an executable parallel program the output of VFCS (i.e. the 11 above mentioned files) has to be transferred to the target machine. Then the build process may be initiated by typing

`source vfbuild.`*project*

(or simply `source vfbuild` in case that no project name has been selected).

Depending on the chosen execution model[6] (*host-node model* or *node-only model*) one or two executable programs are generated. If the node-only model has been selected the build process generates only one executable, *project*`_node`. If the host-node execution model has been chosen the build process generates two executables *project*`_host` and *project*`_node` containing a host program and a node program, respectively.

Execution Model	Executables	Meaning
node-only	*project*`_node`	node program
host-node	*project*`_host`	host program
	project`_node`	node program

5.4.3 Runtime Environment

Running the buildfile sets a couple of runtime environment variables needed by the VFCS runtime system. They are all prefixed with `RMS_` (RMS stands for resource management system).

The meaning of the variables is the following:

Name	Possible Values	Meaning
`RMS_NNODES`	{1, 2, 3, ...}	# of node processors
`RMS_IOMODE`	{HOST_NODE, NODE_ONLY}	execution model
`RMS_ARCH`	{MEIKO, IPSC, PARAGON}	target architecture
`RMS_COMMLIB`	{NX, PARM_5.1, PARM_6.0, MPI}	communication library

If some resource constraints are not met the VFCS runtime system is not able to start the program correctly and aborts with an error message (e.g. appropriate number of processors not available). Note, that on the Meiko CS-2 an appropriate partition has to be selected by the user before executing the parallel program. This is done by setting the `RMS_PARTITION` environment variable. The availability of partitions can be checked with the `rinfo` command.

[6]Note that the *host-node* execution model is not available for MPI.

5.4.4 Executing the Parallel Application

The procedure to run the parallel program is dependent on the execution model. In a pure SPMD model (node-only model) the program is started via a system call e.g.

*load project*_`node`

on the INTEL iPSC and Paragon, or

*prun project*_`node`

on the Meiko CS-2.

If the host-node model has been chosen, it is sufficient to call the host program by name, i.e. simply type

*project*_`host`

on the command-line.

5.4.5 Runtime Environment for Performance Measuring with VFPMS

At each invocation of the measuring routine `TIMEAS`[7] a time-stamp is written to a measuring buffer. The buffer size can be controlled by the environment variable `RMS_TBUFSIZE`. Setting `RMS_TBUFSIZE` to n means that $n \times 1000$ measurings can be processed. The default value for `RMS_TBUFSIZE` is 10.[8] If the buffer is not sufficiently large for the number of measurings the error message *"error too much measuring data, increase RMS_TBUFSIZE!"* is displayed.

The accuracy of the value returned by the system dependent time measuring function[9] can be controlled by the environment variable `RMS_TPRECISION`. To set the precision to an accuracy of 10^{-n} seconds, the variable should be set to a value of n, i.e. 7 for 100 ns.

The format of the time value is determined by the environment variable `RMS_TFORMAT`. If the user wants the time values to be output in 10^{-n} seconds, `RMS_TFORMAT` should be set to n. The default value for both `RMS_TPRECISION` and `RMS_TFORMAT` is 7.[10]

Examples

In the following example we suppose that the system dependent measuring routine yielded the value 0.123456789.

Accuracy	RMS_TPRECISION	Format	RMS_TFORMAT	Output
$1\mu s$	6	s	0	0.123456
$1ms$	3	ms	3	123
$100ns$	7	μs	6	123456.7

[7] a call to `TIMEAS` corresponds to a measuring point calculated automatically by VFPMS (see section 5.3)

[8] i.e. the buffer can contain 10000 time-stamps.

[9] e.g. the measuring function `dclock()` on the iPSC 860 yields double precision values with 100 ns accuracy. Please refer to your operating system manual.

[10] this corresponds to the $100ns$ resolution of `dclock()` on the iPSC.

6 Conclusion

6.1 Summary

Within the first year of the project VFCS has been extended to provide support for many HPF and HPF+ language features. As far as the input language HPF is concerned, regular block distributions and non-standard distributions like `GEN_BLOCK`, `INDIRECT`, and distribution to processor subsets, which also have been accepted for the Approved Extensions of HPF-2, have been implemented. Moreover, most of the HPF alignment features have been realized.

The HPF `INDEPENDENT` directive with the `NEW` clause, `REDUCTION` clause, and an HPF+ extensions for the specification of work distribution has been implemented. Purest procedures that exclusively work on data private to a processor and therefore do not induce any communication may be called from `INDEPENDENT` loops. These language features have turned out to be crucial for all HPF+ benchmark kernels.

The parallelization techniques of VFCS have been significantly extended and improved. In particular the inspector-executor strategy for the parallelization of irregular loops has been significantly improved and integrated with the overlap strategy of VFCS.

The four HPF+ benchmark kernels have been successfully parallelized with the extended version of VFCS. Comparisons with commercial HPF-1 compilers have shown that the language features and compilation techniques provided by VFCS are crucial for handling a broad range of irregular applications.

6.2 Future Work

Additional Language Support

Experiments with the kernels parallelized by VFCS indicate that additional language and implementation support is needed to reduce the overhead implied by the runtime preprocessing of irregular loops. In particular, language and compiler support for the reuse of communication schedules (which are determined in the inspector phase) will have a significant impact on the performance results of the generated parallel programs. A first language proposal that provides support for communication schedule reuse has been worked out and will be integrated in the next HPF+ language definition.

Generalized Parallelization Strategy

Currently the inspector-executor parallelization strategy as implemented in VFCS has the restriction that only arrays with a single distributed dimension can be handled. This restriction is due to the PARTI library which only provides support for one distributed dimension. For the next version of the benchmark kernels it will be necessary to provide support for an arbitrary number of distributed dimensions. Also, the need for allocatable arrays and run-time defined distributions has been identified. A generalized compilation strategy that takes into account these requirements has already been specified and will be implemented within the next project phase. The run-time support system of VFCS, in particular those components that handle irregular loops, will be generalized accordingly. This also requires a redesign of those parts which currently are based on the PARTI library.

Extended Support for Fortran 90

Although VFCS already provides some support for Fortran 90, in particular array assignment statements, a number of important language features can not yet be handled. Support for Fortran 90 features like allocatable arrays, derived types, pointers and procedure interfaces require a non-trivial generalization of the internal representation and the available parallelization strategies. The groundwork for this task has already been carried out by partially integrating a full Fortran 90 / HPF frontend with VFCS. This frontend, which has been developed by GMD Berlin and which is based on the Cocktail compiler construction toolbox, will serve as the basis for the development of a general parallelization strategy for HPF+. An unparser which generates from the internal representation Fortran 90 code has been implemented. Moreover, a library based on the pattern-matching tool PUMA for performing program transformations has been designed and is currently implemented.

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. Fritzson, 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, et.al. Vienna Fortran Compilation System. Version 1.2. User's Guide, Institute for Software Technology and Parallel Systems, February 1996.

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

[8] 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.

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

[10] 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.

[11] T. Fahringer and H. Zima, A Static Parameter based Performance Prediction Tool for Parallel Programs Invited Paper, *In Proc. of the 7th ACM International Conference on Supercomputing*,Tokyo, Japan, July 1993

[12] T. Fahringer, The Weight Finder, An Advanced Profiler for Fortran Programs, In *Automatic Parallelization, New Approaches to Code Generation, Data Distribution, and Performance Prediction*, Vieweg Advanced Studies in Computer Science, Verlag Vieweg, Wiesbaden, Germany, March 1993.

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

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

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

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

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

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

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

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

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

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

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

[24] 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 VFCS 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*
specification-directive	**is**	*processors-directive*
	or	*align-directive*
	or	*distribute-directive*
	or	*inherit-directive*
	or	*combined-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 *hpf-directive-line* follows the rules of Fortran fixed form (3.3.2.1) comment lines.

Note that VFCS currently does not support HPF *combined directives*.

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* `)`

Constraint: The number of processors has to be fixed at compile time.

Constraint: Procedure local processor arrays must not be declared. (The processor arrays of the main program are visible in all subroutines.)

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	`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: An *object-name* mentioned as a *distributee* may not appear as an *alignee* in an `ALIGN` directive.

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 ALIGN Directive

align-directive	**is**	`ALIGN` *alignee align-directive-stuff*
align-directive-stuff	**is**	(*align-source-list*) *align-with-clause*
align-attribute-stuff	**is**	[(*align-source-list*)] *align-with-clause*
alignee	**is**	*object-name*
align-source	**is**	:
	or	*
	or	*align-dummy*
align-dummy	**is**	*scalar-int-variable*

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

Constraint: The *align-source-list* (and its surrounding parentheses) must be omitted if the *alignee* is scalar. (In some cases this will preclude the use of the statement form of the directive.)

Constraint: If the *align-source-list* is present, its length must equal the rank of the alignee.

Constraint: An *align-dummy* must be a named variable.

align-with-clause	**is**	`WITH` *align-spec*
align-spec	**is**	*align-target* [(*align-subscript-list*)]
align-target	**is**	*object-name*
align-subscript	**is**	*int-expr*
	or	*

Constraint: Each *align-subscript* may contain at most one occurrence of an *align-dummy*.

Constraint: Each *int-expr* that is used as an *align-subscript* must be a linear expression in at most one *align-dummy*.

A.5 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 `ALIGN` attribute.

A.6 The INDEPENDENT Directive

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

Constraint: The first non-comment line following an *independent-directive* must be a *do-stmt.*
item 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.7 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*.