Vienna Fortran Compilation System

Version 1.2

User's Guide

**S. Benkner, S. Andel, R. Blasko, P. Brezany, A. Celic,
B.M. Chapman, M. Egg, T. Fahringer, J. Hulman,
E. Kelc, E. Mehofer, H. Moritsch, M. Paul, K. Sanjari,
V. Sipkova, B. Velkov, B. Wender, H.P. Zima**

*Institute for Software Technology and Parallel Systems,
University of Vienna,
Liechtensteinstrasse 22, A-1090 VIENNA, AUSTRIA*

February, 1996

Contents

Preface

This Users's Manual describes the Vienna Fortran Compilation System (VFCS 1.2), a parallelization system for distributed-memory multiprocessing systems. The VFCS performs a source-to-source translation from **Vienna Fortran** or **HPF** to explicitly parallel Message Passing Fortran.

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 menu system based on the X11-MOTIF user interface.

Moreover, VFCS provides integrated support for profiling, performance prediction and performance measurement.

Audience Description

This manual is a VFCS user's guide only. Users are expected to have a basic knowledge of data parallel programming, and some experience in Fortran programming. VFCS is available on several systems and generates output for various target architectures. It is assumed that users are familiar with the basic commands on these systems.

Organization

This *User's Guide* is structured into the following chapters and appendices:

Chapter 1, *Introduction*, gives an overview of the *VFCS* and describes the major compilation techniques.

Chapter 2, *VFCS Command Line Compiler*, describes how to use the system in automatic mode as a command line compiler.

Chapter 3, *VFCS Interactive Compiler*, describes how to use VFCS as an interactive parallelization tool, and how the user may influence the process of restructuring a sequential program into a parallel program.

Chapter 4, *VFCS Target Code Generation*, lists the available target architectures, describes the corresponding runtime libraries, and shows how to execute and measure the generated parallel program.

Chapter 5, *VFCS Analysis and Transformation Tools*, describes the features of the integrated program analysis tool and program transformation tool.

Chapter 6, *VFCS Performance Tools* , describes the *wfprof*, an integrated tool for sequential profiling, which can be used to determine the performance critical sections of a sequential program. Moreover, this chapter describes the use of an integrated parameter-based performance prediction tool that assists the user in fine-tuning the parallel program, and an tool for instrumenting the parallel program for performance analysis.

Appendix A, *Input Languages*, describes the available Vienna Fortran and HPF language features and the limitations of the system.

Appendix B, *Intermediate Language*, describes the semantics of intermediate language constructs used to display information during interactive parallelization.

Appendix C, *Menu Options*, lists the menu entries provided within the VFCS windows.

Appendix D, *Driver Directives*, describes a set of directives which may be used to customize the command-line compiler to specific requirements.

Appendix E, *Installation*, describes the hardware and software requirements for installing VFCS on various different platforms.

Hardware and Software Constraints

VFCS is available for a number of host systems and produces as output parallel message passing programs for a variety of distributed-memory multiprocessors. Details concerning installation, hardware requirements and system availability can be found in Appendix E.

VFCS may be installed for a single user or for a number of users in a workstation network. VFCS generates output for a number of different target architectures using several different parallel communications libraries.

Related Documents

The following documents contain additional information in the context of Fortran and data parallel programming:

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

2. ISO. Fortran 90 Standard, May 1991, ISO/IEC 1539 :1991 (E)

3. *Vienna Fortran - A Language Specification, Version 1.1* Technical Report ACPC/TR 92-4, Austrian Center for Parallel Computation, March 1992

4. High Performance Fortran Forum. *High Performance Fortran Language Specification. Version 1.1*, November 10, 1994.

Chapter 1

Introduction

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

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 provides 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.

Input languages for VFCS 1.2 are Fortran 77, Vienna Fortran, and a subset of HPF. 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.

The parallelization techniques of VFCS are based upon the *data parallel model of computation* in connection with the *Single-Program-Multiple-Data (SPMD)* paradigm.

1.1 Programming Distributed Memory Parallel Computers

In this Chapter we discuss distributed memory multiprocessors (DMMP) and their dominant programming paradigms.

1.1.1 Distributed Memory Parallel Processors

The most important single difference between distributed memory parallel computers (including Intel's hypercube and the Paragon, the nCUBE, and the Meiko Computing Surface) and other computer architectures is the fact that the memory is physically distributed among the processors; the time required to access a non-local datum may be an order of magnitude higher than the time taken to access locally stored data. This has important consequences for program performance. In particular, the management of data, with the twin goals of both spreading the computational workload and minimizing the delays caused when a processor has to wait for non-local data, becomes of paramount importance.

1.1.2 Single Program Multiple Data Programming Model

The most widely used programming model for distributed memory machines is the data parallel *single program multiple data* (SPMD) model of computation in combination with the *owner computes paradigm*. SPMD programs are characterized by a single thread of control; all processors execute the same program but on different parts of the data space. Programming with the SPMD model can be outlined as follows.

- First, the data arrays of the program are partitioned into pieces each of which is assigned to a processor which becomes the *owner* of the data assigned to it.
- Secondly, the work (i.e. the computations) is distributed according to the specified data distribution usually by applying the *owner computes rule*. The owner computes rule implies that each processor only executes assignments to data elements owned by it.
- Finally, access to non-local data must be implemented by *message passing*, i.e. by inserting appropriate communication statements to send and receive data between processors.

A major characteristic of this style of programming is that the performance of the resulting code depends to a very large extent on the data distribution selected by the programmer. The data distribution determines not only where computation will take place. It is also the main factor in deciding what communication is necessary. The total cost incurred when non-local data is accessed involves not only the actual time taken to send and receive data, but also the time delay when a processor must wait for non-local data, or for other processors to reach a certain position in the code. Note that the performance of a program can no longer be estimated solely by the amount of computation it comprises: extra computation is not necessarily costly, and the communication delay inherent in a particular data distribution could be prohibitive.

1.1.3 Explicit Message Passing Languages

Currently, most scientific and numerical applications on DMPP systems are based on the SPMD model using conventional programming languages like Fortran and C with extensions for explicit message passing. With this approach the user has full control of the underlying hardware but is forced to cope with a number of low level details, which can become surprisingly complex: buffers must be set up, and the programmer must take care to send data as early as possible, and in economical sizes. Several issues arise which do not have their counterpart in sequential programming. New types of errors, such as deadlock and livelock, must be avoided. The programmer must decide when it is advantageous to replicate computations across processors, rather than send data. This results in a programming style which can be likened to assembly programming on a sequential machine. It is tedious, time-consuming and error prone, and has led to particularly slow software development cycles and, in consequence, high costs for software.

1.1.4 Data Parallel Languages

Early attempts to develop programming languages that provide appropriate high-level language constructs were based on sequential languages like Fortran or C with extensions for data distribution. They include Blaze [35], extensions to CM Fortran [46], CRAFT [42], Fortran D [16], the Yale Extensions [9], Kali [37], Vienna Fortran [54], and HPF [32].

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

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

1.2 VFCS Data Parallel Language Features

VFCS supports many of the Vienna Fortran and HPF data parallel extensions. Central concepts of Vienna Fortran and HPF are processors and distributions.

- *Processor arrays* may be declared by the user to define the set of *abstract processors* used to execute the program.
- *Distributions* map arrays to non-empty sets of processors.

Currently VFCS provides the following data distribution features:

- *Regular block distributions*:
  A regular block distributions partitions an array dimension to a processor dimension in evenly sized portions.
- *General block distributions:*
  A general block distribution, distributes 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.
- *Distribution to processor subsets:*
  An array may be distributed to a regular processor array section.
- *Distribution of workspaces:*
  Workspaces, which are often used in Fortran 77 programs to simulate dynamic memory allocation may be distributed.
- *Indirect distributions:* Indirect distributions enable the user to specify any arbitrary distribution that does not involve replication by means of a *mapping array*.
- *Distribution by alignment:*
  The distribution of an array may be specified implicitly by aligning it with another (already distributed) array.

`PROCESSORS R(4)`	The `PROCESSORS` declaration introduces `R` as a one-dimensional processor array with four elements. The annotations attached to the declarations specify
`REAL A(12) DIST(BLOCK)`	• a regular *block distribution* for `A`: `BLOCK` distributes an array dimension to a processor dimension in evenly sized portions: array `A` is partitioned into 4 blocks of length 3.
`REAL B(12) DIST(:)`	• (total) *replication* for array `B`. Every processor owns a copy of the whole array.
`REAL C(12) DIST(S_BLOCK((/2,4,4,2/)))`	• a *general block distribution* for `C`: `S_BLOCK` distributes an array dimension in arbitrarily sized portions. Array `C` is partitioned into 4 blocks of lengths 2,4,4, and 2.
`REAL D(12) DIST(BLOCK) TO R(2:3)`	• a block distribution to a *processor array section* for `D`: array `D` is partioned into 2 blocks of length 6 which are mapped to processors `R(2)` and `R(3)`, respectively.
`REAL E(12) DIST(INDIRECT(MAP))`	• an indirect distribution according to the contents of array `MAP`: `A(`i`)` is mapped to processor `MAP(`i`)`, $\forall i, 1 \leq i \leq 12$.
`REAL F(12) ALIGN (F(I) WITH D(I))`	• an alignment for `E`: array element `E(I)` is located on the same processor as element `D(I)`.

Basic Data Distributions

Furthermore, in the context of Vienna Fortran the user may specify the distribution of the iterations of parallel loops by means of an `ON`-clause.

The Vienna Fortran and HPF subsets that are currently accepted by VFCS and differences between the two languages are described in Chapter 2.

1.3 Parallelization Strategy

This section gives an overview of VFCS's parallelization strategy. In order to generate an efficient parallel program VFCS has to manage data locality, parallelism and communication. The efficiency of the parallel program is significantly influenced by the user specified data distribution. Thus the user needs to be aware of some details of the data distribution features of VFCS.

1.3.1 Overview

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.

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. Distribution analysis is performed on an interprocedural basis, i.e. by propagating data distribution information over the call graph of the program, in order to determine the distribution of dummy arguments.

Based on the information obtained after interprocedural distribution analysis, VFCS automatically partitions the control by applying the owner computes rule. This is implemented by associating each assignment statement with a boolean guard called *mask*. The mask ensures that each processor only executes assignments to data owned by it. In a further step, masks are eliminated wherever possible; e.g. by adapting the bounds of loops.

If access to non-local data is necessary appropriate message-passing statements are inserted in the generated code in order to communicate data between the processors. Memory to hold non-local data is automatically allocated and global references are translated into local references. Since reduction of communication costs is crucial in achieving good performance, the communication is optimized where possible, in particular by avoiding unnecessary communication, hoisting communication statements out of loops, and combining them.

1.3.2 Compilation Phases

The translation from Vienna Fortran or HPF to message passing Fortran can be described as a sequence of five phases:

- **Phase 1: Front End**
  The *front end* performs the following three tasks: it transforms the source code into an *internal representation* suitable for further compiler manipulation, it performs the *initial analysis* of the program, and it *normalizes* the code.
- **Phase 2: Standard Transformations**
  This phase transforms the program in such a way that subsequent parallelization transformations will be more effective.
- **Phase 3: Basic parallelization**
  In this phase the following steps are performed to obtain an initial parallel program.
  - *Splitting* divides the program into a *host part* and a *node part*. The host part will, after compilation, be executed on the host computer or a specially designated host node of the target system, by the *host process*. It performs global management tasks, such as requesting resources, loading code, and terminating program execution; it also handles all input/output. On each processor of the target machine, a *node process* will execute the node part to perform the actual computation.
  - *Masking* enforces the owner computes paradigm by associating a boolean guard, called the **mask**, with each statement, in accordance with the ownership implied by the distributions.
  - *Communication insertion* generates for all potentially non-local data accesses high-level communication statements which copy non-local data items to private variables of the processor.
- **Phase 4: Optimization**
  The optimization phase eliminates redundant communication, hoists communication statements out of loops, and combines small messages into larger ones in order to minimize the effects of latency. Overlap analysis detects certain simple regular communication patterns and re-organizes communication based upon this information, and determines the minimum amount of storage required for copies of non-local data. Mask optimization adapts loop bounds such that unnecessary loop iterations are eliminated.
- **Phase 5: Target code generation**
  The final phase creates an optimized parallel message passing Fortran target program. High-level communication statements are implemented by calls to a message passing library.

1.3.3 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 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. If a loop performs a reduction operation, it is substituted by a call to a parallel reduction function.

- `INDEPENDENT` loops
  Vienna Fortran `INDEPENDENT` loops enable the user to assert that the iterations of a loop are independent and can be executed in parallel. Furthermore, the work distribution (i.e. the distribution of loop iterations to processors) may be specified by the user. Parallelization of `INDEPENDENT` loops is implemented based on run-time compilation techniques.

1.4 Modes of Operation

VFCS provides two different modes of operation to the user: an *automatic mode* and an *interactive mode*.

1.4.1 Automatic Mode

When used in *automatic mode*, as a command-line compiler, VFCS provides a set of easy-to-use directives for the creation of *driver* files which are used in addition to command-line options to customize the system to the user's needs. A driver file controls the sequence of transformations that are applied during the transformation process. Default driver files for different parallel architectures are provided. Furthermore, driver files may be generated automatically by protocoling the actions taken during an interactive parallelization session. Using VFCS as a command-line compiler is described in Chapter 3.

1.4.2 Interactive Mode

When used in *interactive mode*, the system offers a set of analysis services and a transformation catalog which allow the user to guide the system through the parallelization process. The interactive mode is supported by a hierarchically structured menu system based on the X11-MOTIF user interface. Using VFCS as an interactive parallelization tool is described in Chapter 4.

1.5 System Structure

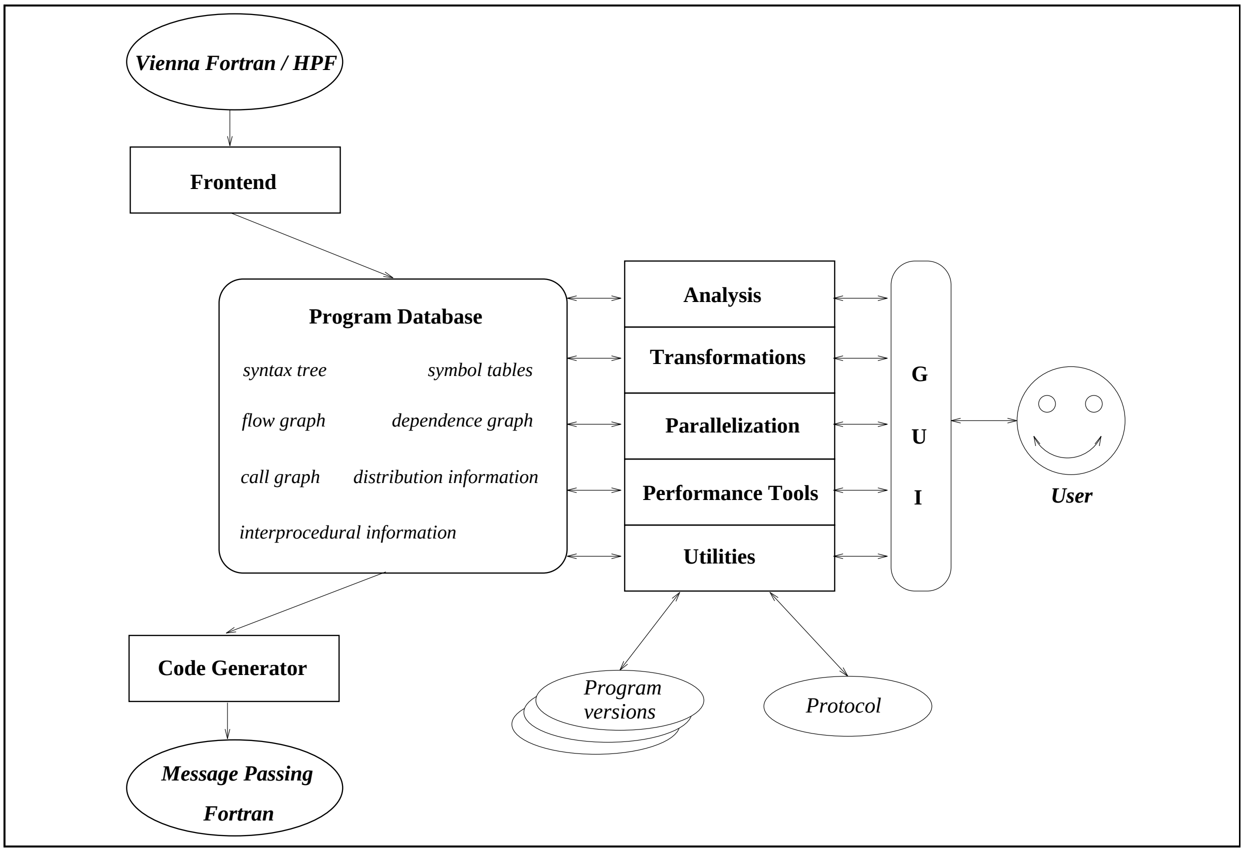


Figure 1.1: The Structure of VFCS

Figure 1.1 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.

Chapter 2

VFCS Input Languages

VFCS accepts as input languages Fortran 77 with Fortran 90 extensions and subsets of Vienna Fortran and of HPF.

In this section we describe the major data distribution strategies provided by VFCS. The syntax of the Vienna Fortran and HPF data-parallel constructs which are currently accepted by VFCS can be found in Appendix A. For syntactic and semantic details refer to the language specifications.

2.1 Processor Arrays

This version of the system requires that at least one processor array is specified and that the number of the processors is known at compile time. Processor array declarations may currently only appear in the main program unit and are accessible in all other program units. If more than one processor array is declared, the size of all processor arrays must be the same.

The following example shows how to declare processor arrays in Vienna Fortran or HPF. Note that the current implementation establishes a one-to-one correspondence between the elements of different processor arrays by means of the Fortran column major order, i.e. element R2(2,2) and element R(12) in Example 2.1 designate the same processor. Note that currently VFCS permits only one `PROCESSORS` statement in a Vienna Fortran program.

Example 2.1: Processor Array Declarations

```
! Vienna Fortran processor arrays:
   PROGRAM MAIN

   PROCESSORS VR2(10,10) RESHAPE VR(100)
```

```
! HPF processor arrays:
   PROGRAM MAIN

   !HPF$ PROCESSORS HR2(10,10)
   !HPF$ PROCESSORS HR(100)
```

The Vienna Fortran `PROCESSORS` *statement declares* `VR2` *as* primary processor array *and* `VR` *as* secondary processor array*. In HPF such a distinction does not exist.*

In Vienna Fortran the processor array (section) to which a distribution refers may be specified after the keyword `TO`. If no processor reference is specified, the distribution refers to the *primary* processor array. Regular subsets of a processor array may be referenced by means of section subscripts. The dimensions of processor arrays may be *permuted*.

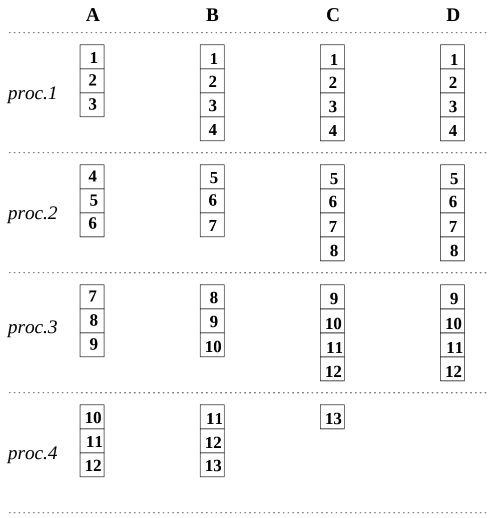
In HPF a distribution specification *must* designate after the keyword `ONTO` the processor array to which the distribution refers. Distribution to processor subsets and permutation of processor array dimensions is not supported by HPF.

2.2 Data Distribution Strategies

VFCS supports several variants of BLOCK distributions, partial replication, distribution to processor subsets, distribution by alignment, inherited distributions, and distribution of Fortran work spaces. In the following the basic distribution features of VFCS are described by means of examples. If necessary, differences between Vienna Fortran and HPF are indicated. If not stated otherwise, it is assumed that a one-dimensional array of size N is distributed with respect to a one-dimensional processor array with P processors. Multidimensional distributions are described in Section 2.2.6.

Example 2.2: Regular Block Distributions

```
      PROCESSORS :: R1(4)

      REAL A(12) DIST(BLOCK) TO R1

      REAL B(13) DIST(BLOCK)

      REAL D(12) DIST(BLOCK(4))

!HPF$ DISTRIBUTE(BLOCK) ONTO R1 :: A(12)

!HPF$ DISTRIBUTE(BLOCK) ONTO R1 :: C(13)

!HPF$ DISTRIBUTE(BLOCK(4)) ONTO R1 :: D(12)
```



2.2.1 Regular Block Distributions

A regular block distribution partitions an array dimension into contiguous blocks. Both Vienna Fortran and HPF provide two forms of block distributions; i.e. `BLOCK` where the size of the blocks is determined by the compiler as shown below, and `BLOCK(M)` where the size of the blocks is specified explicitly. In the case where the array size, N, is not exactly divisible by the number of processors, P, the semantics of the Vienna Fortran `BLOCK` and the HPF `BLOCK` distribution is slightly different. If N is divisible by P, then both `BLOCK` distributions partition the array into P blocks of size N/P.

Vienna Fortran `BLOCK`:

The array is partitioned into P contiguous blocks whose sizes differ by at most 1. The first $r := N \ mod \ P$ blocks have size $\lceil N/P \rceil$ and the remaining $P - r$ blocks have size $\lfloor N/P \rfloor$.

HPF `BLOCK`**:**

The array is partitioned into P blocks whose sizes differ by at most $\lfloor N/P \rfloor$. The first $P-1$ blocks have size $\lceil N/P \rceil$ and the last block contains the remaining elements.

`BLOCK(M)`**:**

This distribution has the same semantics in Vienna Fortran and HPF. The array is divided into $n = \lfloor N/M \rfloor$ contiguous blocks of size M which are mapped to the first n processors. If N is not exactly divisible by M, then the remaining ($N \ mod \ M$) elements are mapped to processor $n+1$. All other processors (if any) do not receive any elements. Note that $M \geq \lceil N/P \rceil$ must be satisfied.

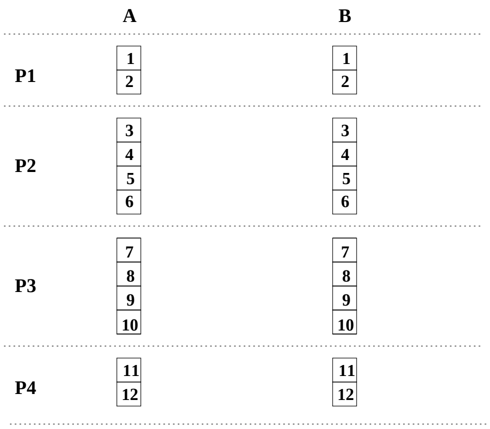
2.2.2 General Block Distributions

A general block distribution enables the user to partition an array dimension into arbitrarily sized blocks. Vienna Fortran provides two forms of general block distributions `B_BLOCK` and `S_BLOCK` both of which take a one-dimensional integer array `G(1:P-1)` as argument. In case of `B_BLOCK`, element `G(I)` specifies the upper bound of the i-th block, and in case of `S_BLOCK` the size of the i-th block.

General block distributions enable the user to influence load balancing. For example, they are suitable for algorithms which involve more computation at the boundaries of the data domain than in the interior. General block distributions are not available in HPF.

Example 2.3: General Block Distributions

```
PROCESSORS :: R1(4)

PARAMETER (BB = (/ 2,6,10 /))
PARAMETER (SB = (/ 2,4,4  /))

REAL A(12) DIST(B_BLOCK(BB))

REAL B(12) DIST(S_BLOCK(SB))
```



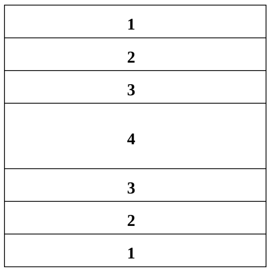
2.2.3 Indirect Distributions

Indirect distributions enable the user to handle effectively irregular codes where array access patterns are dependent on runtime data. An indirect distribution is defined by means of the intrinsic function INDIRECT and a mapping array, and can express any arbitrary distribution that does not involve replication. It partitions an array irregularly according to the contents of the mapping array. In the current implementation only the first dimension of an array may be distributed indirectly. Thus, mapping arrays are currently restricted to one-dimensional integer arrays which must have the same size as the array dimension to be distributed. Each element of the mapping array must designate a valid processor number.

In the current implementation, a `DISTRIBUTE` statement has to be used in order to instantiate an indirect distribution before the corresponding arrays are referenced. Since currently only statically distributed arrays are supported, each indirectly distributed array may be the object of only a single distribute statement. The mapping array, once it has been defined, has to stay invariant during program execution. Moreover, indirectly distributed arrays may only be referenced within `DO INDEPENDENT` loops, and must not be the objects of I/O statements.

Example 2.4: Indirect Distributions

```
PROCESSORS :: R1(4)

INTEGER MAP(10)
REAL    A(10) DIST(INDIRECT(MAP))
....
! code for computing contents of MAP array
....
! instantiation of indirect distribution

DISTRIBUTE A :: (INDIRECT(MAP))
```



Layout of **A**

2.2.4 Replication

To indicate that an array dimension should be replicated, i.e. allocated on each processor in Vienna Fortran the ":" is used whereas in HPF the "*" is used.

2.2.5 Default Distribution

The distribution of arrays which are not explicitly distributed by the user will be determined by the compiler. Currently VFCS replicates scalar and arrays for which no distribution has been specified. This is not the case for dummy arrays: a dummy array will *inherit* the distribution of the actual array by default (see 4.4.2).

2.2.6 Multidimensional Distributions

Multidimensional distributions may be defined by specifying a distribution for each array dimension separately. There are significant differences between multidimensional distributions in Vienna Fortran and HPF. These are discussed below.

Vienna Fortran Multidimensional Distributions

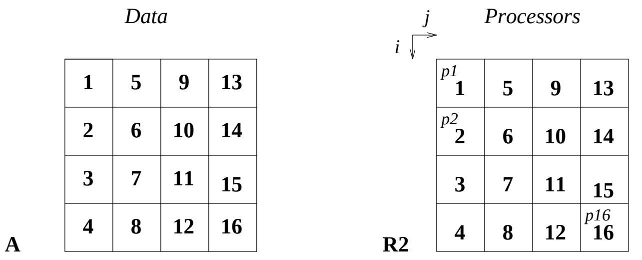
In Vienna Fortran the number of distributed dimensions, denoted by d, must be less than or equal the rank of the processor array (section), denoted by p, to which the distribution refers.

Case 1: $d = p$

> The d distributed dimensions are partitioned with respect to the d processor array dimensions, i.e. dimension i of the array is distributed with respect to dimension i of the processor array, $1 \leq i \leq d$.

Example 2.5: Multidimensional Distributions

```
PROCESSORS :: R2(4,4)

REAL A(N,N) DIST(BLOCK,BLOCK)
```



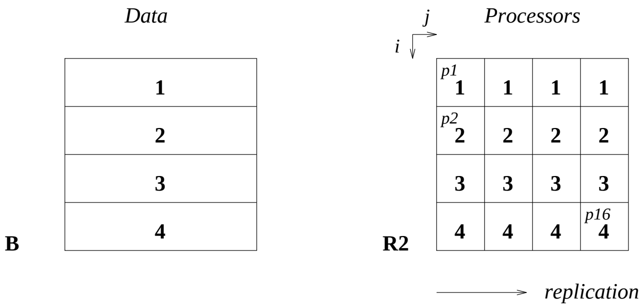
In this example array A *is distributed by* BLOCK *in both dimensions. The resulting* 16 *segments of* A *are mapped to the* 16 *processors of processor array* R2 *and as shown in the figure.*

Case 2: $d < p$

The array is distributed with respect to the first d processor array dimensions, and partially replicated over the remaining $p - d$ processor array dimensions.

Example 2.6: Partial Replication

```
PROCESSORS :: R2(4,4)

REAL B(N,N) DIST(BLOCK,:)
```



In this example array B *is distributed by* BLOCK *in the first dimension while the second dimension stays undistributed. The resulting 4 segments of* B *are mapped to the first dimension of processor array* R2 *and replicated over the second dimension of* R2 *because the distribution rank of array* B *is 1.*

Case 3: $d > p$

Such a distribution is illegal and rejected by the system.

Furthermore, in Vienna Fortran the dimensions of a processor array may be permuted by supplying a dimension permutation after the processor reference.

Multidimensional Distributions in HPF

In HPF the number of distributed dimensions of an array must always be equal to the rank of the processor array. Partial replication of data and permutation of processor dimensions can only be achieved by means of alignment.

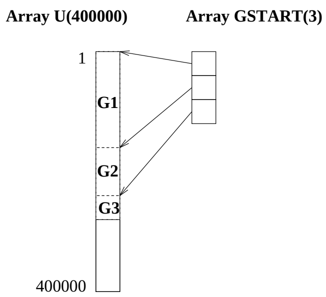
2.2.7 Distribution of Workspaces

Many applications written in FORTRAN 77 use workspaces to circumvent the lack of dynamic storage allocation. A workspace is a large array reserving storage for several different arrays which are mapped into the workspace. The workspace is generally accessed by means of dummy arrays which are bound at run-time to contiguous sections of the workspace.

VFCS supports the parallelization of applications that use workspaces. The user may declare an array to be a workspace, describe the mapping of *virtual arrays* into the workspace and distribute the virtual arrays according to the requirements of the program.

Example 2.7: Distribution of Workspaces

```
WORKSPACE MG :: G1(129,129) DIST(BLOCK,BLOCK),
                G2(65,65)   DIST(BLOCK,BLOCK),
                G3(33,33)   DIST(BLOCK,BLOCK)

REAL U(40000) DIST(WORKSPACE=MG)

DO K=1,3
   CALL RELAX(...,U(GSTART(K)),GDIM(K),...)
....

SUBROUTINE RELAX(...,G,N,...)
REAL G(N,N) DIST(*)
....
```



Let `REAL U(40000)` *be the work array and* `G1(129,129)`, `G2(65,65)` *and* `G3(33,33)` *three grids forming the grid hierarchy in a multi-grid application. The* `WORKSPACE` *statement enables the user to define a workspace* `MG` *consisting of several virtual arrays.* `MG` *may then be used to distribute the workspace array* `U`. *The virtual arrays* `G1,G2` *and* `G3` *are accessed via dummy arrays which are bound at run time to single grids.* `GSTART` *is used to keep start position of the virtual arrays in the workspace.* `DIST(*)` *indicates that the dummy array* `G` *inherits the distribution from the actual array.*

The parallelization of applications that use workspaces requires user interaction during the distribution analysis phase as explained in Section 4.4.2.5.

Note, too, that in the current implementation arrays which are distributed using the workspace concept must not appear in I/O statements.

2.2.8 Alignment

Alignment is a method to define the distribution of an array B implicitly by specifying for each element that it should be distributed to the same processor(s) as a certain element (or section) of an array A. Note that there are differences between the alignment specifications of Vienna Fortran and HPF. In particular, the ”:” and ”*” are used with different meanings. For syntactic and semantic details refer to the language specifications. The alignment features as currently supported by VFCS are listed in Example 2.8.

Example 2.8: Alignment

```
REAL S(100,100) DIST(BLOCK,BLOCK)
REAL S1(100)    DIST(BLOCK)

! Perfect Alignment
      REAL A(100,100) ALIGN(A(I,J) WITH S(I,J))          ! VF77
      REAL, ALIGNED(S) :: A(100,100)                      ! VF90
!HPF$ ALIGN A(I,J) WITH S(I,J)                            ! HPF

! Offsets
      REAL B(0:99,0:99) ALIGN(B(I,J) WITH S(I+1,J+1))    ! VF77
      REAL, ALIGNED(S) :: B(0:99,0:99)                    ! VF90
!HPF$ ALIGN WITH S :: B(0:99,0:99)                        ! HPF

! Strides
      REAL C(50,50) ALIGN(C(I,J) WITH S(2*I,2*J-1))      ! VF77
      REAL, ALIGNED(S(2::2,::2)) :: C(50,50)              ! VF90
!HPF$ ALIGN C(I,J) WITH S(2*I,2*J-1)  :: C(50,50)         ! HPF

! Permutation of Axes
      REAL D(100,100) ALIGN(D(I,J) WITH S(J,I))          ! VF77
      REAL, ALIGNED(D(I,J) WITH S(J,I)) :: D(100,100)    ! VF90
!HPF$ ALIGN(D(I,J) WITH S(J,I)) :: D(100,100)            ! HPF
```

2.3 Parallel Loops

Whereas the work distribution is usually derived automatically from the data distribution, the distribution of loop iterations in the context of `DO INDEPENDENT` loops may be specified explicitly by the user by means of an `ON`*clause*.

By means of the (optional) `ON`*clause* the loop iterations may be distributed

- to specific processors,
- to the owner of an array element,
- or in a block
- or cyclic fashion.

If no `ON`*clause* is specified, the system chooses a default work distribution.

`DO INDEPENDENT` loops are particularly useful in situations where the compiler cannot detect the absence of loop carried dependencies, for example if subscripted variables are used in subscript positions. Using a Vienna Fortran `DO INDEPENDENT` loop, the user asserts that the different instantiations of the loop body are independent and can be logically executed in parallel. To ensure such independence, `DO INDEPENDENT` loops are not allowed to have any loop-carried dependences (except accumulation type dependences). That is, a data item written in one iteration cannot be read or written in any other iteration. Given this restriction, the program can then execute the iterations in any order. There is an implied synchronization at the beginning and the end of a `DO INDEPENDENT` loop: all processors executing the iterations of the loop are synchronized before the start of the loop, and similarly, they are synchronized at the end of the loop before continuing with the execution of the statements after the loop.

To permit global operations to be performed by a `DO INDEPENDENT` loop `REDUCE` statements are provided.

Example 2.9: Vienna Fortran `DO INDEPENDENT` loops

```
PROCESSORS R(10)
REAL A(100), B(100) DIST(BLOCK)

DO INDEPENDENT I=1,100 ON R(I/10)
   A(I) = A(B(I))
END DO INDEPENDENT

DO INDEPENDENT I=1,100 ON OWNER(A(I))
   A(I) = A(B(I))
END DO INDEPENDENT

DO INDEPENDENT I=1,100 BLOCK
   REDUCE(MULT,A(I),A(B(I))
END DO INDEPENDENT

DO INDEPENDENT I=1,100 BLOCK
   A(I) = A(B(I))
END DO INDEPENDENT

DO INDEPENDENT I=1,100 CYCLIC
   A(I) = A(B(I))
END DO INDEPENDENT

DO INDEPENDENT I=1,100 BLOCK
   REDUCE(MULT,S,A(B(I))
END DO INDEPENDENT
```

2.4 Restrictions

Not all distributions which can be specified for the arrays in a program are necessarily valid. Here, we describe the reasons why certain distributions may be rejected by the system.

Storage and Sequence Association

An array can only be distributed (i.e. other than replicated) if all other arrays sharing storage with it use exactly the same storage and have an element type with the same type length. Sharing of storage is relatively frequent in Fortran 77 codes, since it is achieved by the usage of `COMMON` and `EQUIVALENCE`. The system will reject your data distribution with a corresponding error message if one of the above conditions is not met. This may mean that you should consider modifying `COMMON` blocks in the program.

Restrictions on Dummy Arguments

It is not permitted for a dummy argument of a subroutine to be associated with both a virtual array of a work space and a declared Fortran array at run time. If this occurs and the system cannot find a way to remove the conflict, it will also report an error. Furthermore, in the current implementation, dummy arrays always inherit their distribution from the actual array.

Note, too, that you will sometimes be asked to provide information on the arrays which may be associated with a dummy argument at run time. This topic is discussed in Section 4.4.2.

A detailed description of the Vienna Fortran and HPF subset that is currently accepted by VFCS can be found in the Appendix.

2.5 Preparing Your Program for Parallelization

Before you submit a program to VFCS, you should check that it adheres to the current system limitations described below.

- VFCS should not be used to debug your Fortran code. The system rejects any erroneous program, but sometimes in a very unsophisticated manner. In such a case, check the program with your Fortran compiler and -ANSI switch before starting VFCS. (Note, that in case of a Vienna Fortran program you will have to remove the data distribution extensions first.)
- VFCS currently has strict limitations concerning the size of programs. It can handle codes with ca. 100 subroutines, although the exact number will vary according to the I/O behavior of your program. If a memory fault occurs when applying a transformation to a relatively large subroutine, then you should split the subroutine (or function) into several smaller ones and resubmit your program. If a symbol table overflows, you will receive an appropriate message. Again, you will need to modify your source program to get smaller subroutines (in this case, with fewer variables).
- VFCS accepts currently only a restricted form of I/O statements. In particular the system does not accept namelist I/O and I/O statements with an implied `DO` list. This construct complicates the splitting phase and has been omitted from the Fortran syntax. If this occurs in your program, you should rewrite the corresponding passages using explicit DO loops.

  FORTRAN 77 `READ` and `WRITE` statements must not contain:

  - namelists
  - io-implied-do lists
  - replicated dummy arrays
  - distributed adjustable dummy arrays
  - distributed dummy arrays with non-standard distributions
  - character variables
- VFCS does not accept program if a subroutine passed as an argument to another subroutine performs I/O.
- VFCS currently does not handle `DATA` statements referencing distributed arrays. These should be substituted by corresponding assignments.
- VFCS does not accept all forms of `FORMAT` statements.
- The system currently does not handle all forms of statements assigning to and from `CHARACTER` variables.
- The system does not always correctly process I/O where array variables are individually read/written in a loop with a negative stride.
- VFCS applies a relatively aggressive dead code elimination. You should not attempt to test code by removing I/O, since the system will then "eliminate" the entire program! If you do write a small program for test purposes and code appears to be missing in the output, include some extra write statements to output the variables on the left-hand sides of the missing statements.

Chapter 3

VFCS Command-Line Compiler

This chapter describes the use of VFCS in automatic mode as a command-line compiler. It describes the features which are provided, explains how VFCS is invoked, which command-line options are available, and shows how the user may control the sequence of transformations by means of *driver files*.

3.1 Overview

To support the automatic mode, VFCS provides

- a set of directives, the *driver directives*, to specify commands for the transformation of programs
- an *interpreter* to execute these commands
- a *system component* to protocol the actions during an interactive parallelization session.

Most of the commands of VFCS can be formulated using the *driver directives*. This includes

- commands which are applied to a whole unit (for example, vector code generation)
- commands that select a language construct (e.g. a statement, an identifier, a subroutine call or a loop)
- commands which are applied to the selected construct.

The *driver directives* are described in the Appendix.

3.2 Invoking VFCS

Whenever VFCS is invoked in automatic mode, the actions taken by the system are controlled by a driver file. Depending on the target architecture different default driver-files are provided. The user may create his own driver files by means of a text editor or by protocoling an interactive parallelization session.

If VFCS is invoked with a default driver file, it usually performs the following actions to transform a Vienna Fortran or HPF program into an SPMD message passing program:

- Syntactic and semantic checks
- Processing of data distribution annotations
- Generation of a Fortran 77 SPMD message passing program

3.2.1 VFCS Command-Line Syntax

The VFCS command-line syntax is

vfcs [*options*] [*filename*]...

Where:

options	is zero or more command-line options. For a complete description of the options refer to Section 3.2.2
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 as described in Chapter 4. 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.

3.2.2 VFCS Command-Line Options

In this section we describe the syntax and operation of the command-line options. The options are arranged in alphabetical order. Options may not be grouped together, but may be truncated as long as they are unambiguous.

The following notation is used:

[*item*]	square brackets indicate that the enclosed item is optional
...	horizontal ellipses indicate that zero or more instances of the preceding item are valid.

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*	name the generated message passing source file(s) *project_type*.f; depending on the target architecture, the message passing program may be contained in more than one file; *type* may be *host, hostsys, node, nodesys,* or *init*; (see 3.3.2)
-*v*	set verbose output for automatic mode

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

3.2.3 VFCS Driver Files

VFCS, when invoked in automatic mode, is executed under 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 Appendix. Furthermore, driver files may be generated automatically by protocoling an interactive parallelization session as described in Section 4.9.2.

If the user wants to use her/his own driver files, the corresponding file has to be specified as a command-line option by means of the *-a* or *-b* flag.

3.3 Filename Conventions

VFCS uses the filenames that you specify at the command-line to find and create input and output files. This section describes the input and output filenames for the phases of the compilation process.

3.3.1 Input Files

VFCS accepts as input Fortran 77, Vienna Fortran, or HPF source files. The system determines the type of each input file by examining the filename extension according to the following conventions:

filename.f	indicates a Fortran 77 source file
filename.vf	indicates a Vienna Fortran source file
filename.hpf	indicates a HPF source file
filename.drv	indicates a driver file
filename.pro	indicates a driver file created by the protocol facility

3.3.2 Output Files

VFCS creates several files in your 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.

The parallel program is contained in the following files:

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 your 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, 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*.

3.3.3 Other Files

The *units directory*, *./units*, is created automatically in the working directory (i.e. the directory where you start VFCS) to hold internal program information needed during the compilation process.

A configuration file, *.vfcs*, is created automatically in the the user's home directory.

A file containing the default parameters for target code generation, *.besettings*, is created in the working directory.

Note: These files are for internal use only, and must not be modified by the user.

3.4 Target Compilation and Execution of Parallelized Program

The generated code for the parallelized application is saved in the files *host.f*, *node.f*, and *init.**. These files have to be compiled with the target compiler of the parallel machine in order to obtain an executable program. The *init.** files are used to initialize the node codes and are thus also required at run time.

For compiling and executing the parallel SPMD Fortran program a Makefile, customized for the selected target architecture, is generated. This is described in detail in Chapter 5.

Chapter 4

VFCS Interactive Compiler

This chapter describes the use of VFCS in interactive mode. In this mode VFCS provides a set of analysis services and a transformation catalog which allow 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.

Although VFCS was primarily conceived as a system for converting Fortran code augmented by a data distribution into parallel message-passing Fortran code, it has a broader functionality. It is also possible to use this system to apply program transformations, such as *loop unrolling*, to sequential Fortran code. Many of the transformations which are used to improve the performance of parallel programs are useful for performance tuning of sequential code: the analysis required to determine the applicability of a transformation is the same in both cases. In this version of the VFCS manual, we do not describe sequential processing of files further. If a sequential file is being transformed, many of the options in the menus cannot be validly applied, and hence the system provides a restricted functionality. This functionality includes the viewing and selection of program units, and the application of program transformations.

The integrated program analysis and program transformation tools are described in Chapter 6.

4.1 Invoking Interactive VFCS

The VFCS is invoked in interactive mode by either omitting the source file or by specifying the *-i* option at the command-line (See Section 3.2).

VFCS can is started in interactive mode as follows:

vfcs

After VFCS has been started, the interactive VFCS window as shown in Figure 4.1 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 4.2. A detailed description of the available menu items can be found in Appendix C.

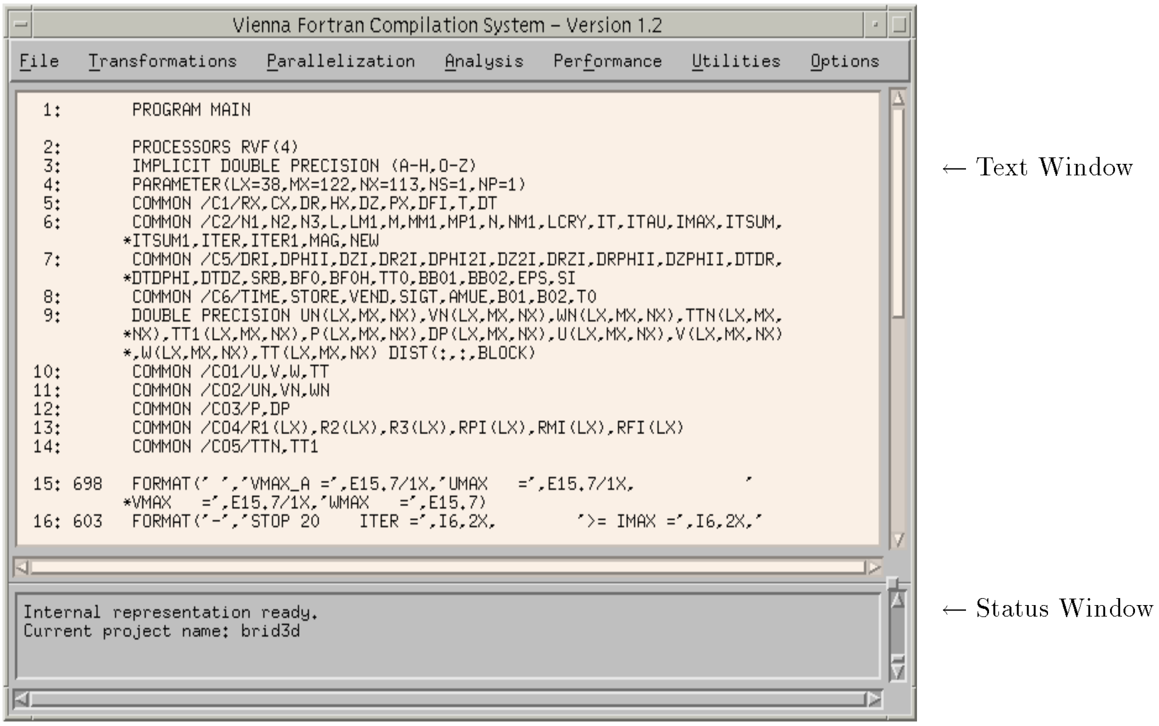


Figure 4.1: VFCS Main Window

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	Mark Parallel Loops	Communication Optimization	Procedure Calls		History	
Exit	Inline Expansion	Mask Optimization	Common Blocks		Protocol On	
	Loop Distribution	Code Generation	Data Distribution		Protocol Off	
	Loop Interchange		Overlap			
	Loop Unrolling					
	Unroll Jam					
	Mask Propagation					
	Expr. Simplification					
	Dead Code Elimination					
	Create Vector Code					
	Destroy Vector Code					

Figure 4.2: VFCS 1.2 - Menu Structure

4.2 Main Steps in Parallelization

Before we discuss the features of the system and the individual menu items in more detail, we outline the steps taken to parallelize a Vienna Fortran program.
These are the steps which **must** be performed in order to parallelize a program; all other possible transformations have the purpose of improving the resulting code. In subsequent sections, we more fully explore the capabilities of the system and explain the menu entries in more detail; an example parallelization session is given. Further details on the available transformation and analysis services are the subject of subsequent chapters.

The following steps have to be performed in order to parallelize a Vienna Fortran or HPF program. If the source program does not contain data distributions, these have to be added before step 2 as described in Section 4.3.3.

Step 1: Load Source Files

Reads in the source files of a program and performs scanning, parsing, program normalization, semantic analysis and generates an internal representation of the program.

Step 2: Basic Parallelization

Performs distribution analysis, overlap analysis, loop adaptation, and generates an initial parallel program referred to as defining parallel program.

Step 3: Optimization

Overlap analysis detects certain simple regular communication patterns and determines the minimum amount of storage required for copies of non-local data.

Communication optimization eliminates redundant communication, hoists communication statements out of loops wherever possible, and combines small messages into larger ones.

Mask optimization adapts loop bounds according to the distribution of arrays defined in loops.

Step 5: Code Generation

Generation of target architecture specific Fortran code.

Step 6: Save Target Files The generated parallel program is saved to file by according to the conventions described in Section 3.3.2

Optional Standard 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, reduction recognition and transformation of `DO` loops into `DO INDEPENDENT` loops.

4.3 Loading Source Programs

The first step in parallelizing an application consists of selecting the files which make up the complete program. VFCS 1.2 will only operate correctly on entire programs.

To select your source files, you must enter the *File* menu and then select the menu item *Load Source Files*. The file selection box then appears on the screen so that you may locate and select the program files.

After file selection, the system automatically initiates the program analysis by reading each of the source files in turn and constructing an internal representation of the program. This phase is called *front-end processing* and consists of traditional compiler analysis such as scanning, parsing and semantical analysis as well as program normalization.

It is advisable to run the sequential source file through a sequential Fortran compiler first to ensure that there are no programming errors, since VFCS does not yet provide full information on all of the errors it detects. There are a small number of I/O constructs which are not accepted by the current version of the system; these are listed in Chapter 2.5. If you do not remove these from your source, the system will report a program error during the initial processing.

4.3.1 Program Units

After the program has been loaded as outlined above, VFCS represents the program as a set of units. A *unit* is either the main program, a subroutine or a function. The list of all available units can be obtained by selecting a menu entry from the *Utilities* menu. The unit that is currently displayed in the VFCS program window is called *actual unit*.

Switching to an other program unit can be done by selecting the *Load Unit* entry form the *Units* submenu of the *Utilities* menu, or via the call graph by double-clicking at the corresponding node. The call graph of the program is displayed after selecting the *Call Graph* entry from the *Analysis* menu.

4.3.2 Program Normalization and Standard Transformations

Program normalization has the purpose of generating a standard form of the program in order to simplify subsequent compiler operations. It is performed automatically whenever a program is loaded. Normalization transformations typically include loop normalization, if conversion and subscript standardization. Array declarations, for example, are transformed in such a way that the lower bound in each dimensions is 1, and all corresponding references are transformed accordingly. Loops are normalized such that the step size becomes 1. Subscript expressions are standardized in order to simplify data dependence analysis. Furthermore standard transformations like constant propagation and dead code elimination are applied.

Additional transformations, such as induction variable substitution, different loop transformations and others, may be selected by the user in order to adapt the program in such a way that subsequent parallelization transformations yield better results.

Details concerning these transformations can be found in Section 6.2.

4.3.3 Interactive Data Distribution

For Vienna Fortran or HPF programs the data distributions is usually specified directly in the source code. However, if a pure Fortran program is loaded, the data distribution has to be performed interactively by the user, before parallelization. VFCS provides the *Interactive Data Distribution Window* to enable the user to specify or change the distribution of arrays interactively. It is invoked by selecting the menu entry *Data Distribution* from the *Parallelization* menu.

Whenever the *Interactive Data Distribution Window* is invoked, all distribution specifications of the entire program are displayed. The user then can modify or delete existing distributions or define new distributions. Furthermore data distributions may be read from or saved to a file.

The interactive data distribution window is described in detail in Section C.3.1 of the Appendix.

4.4 Basic Parallelization

At the beginning of this phase, a program has been analyzed, and normalized. Moreover, the data distribution for that program has been completely specified (either as a part of a Vienna Fortran or HPF program, or interactively, by using the data distribution window). Furthermore, if necessary, standard transformations as described in the previous section should have been applied e.g. induction variable substitution, scalar forward substitution.

Basic parallelization is performed by applying the following steps to the node program:

1. *Splitting:*
   Program splitting generates a host part and a node part of the program. It is described in Section 4.4.1.

2. *Distribution analysis:*
   Distribution analysis automatically determines the distribution of dummy arguments as described in Section 4.4.2. However, there may be situations where the system is unable to determine certain properties of the distribution of dummy arguments and thus the user may be asked for additional information.

3. *Masking:*
   Masking ensures that each processor executes only assignments to those elements of a distributed array that are owned by it.

4. *Communication insertion:*
   If a processor needs to access elements which are not owned by it, appropriate communication statements are automatically generated.

All these 4 steps are automatically performed by the system after selection of the *Basic Parallelization* menu entry. The program obtained after these steps will be referred to as the *defining parallel program.*
Note: `DO INDEPENDENT` loops are not transformed during *basic parallelization.*

4.4.1 Splitting

In this phase the sequential source program is split into a *host program* and a *node program.* The host program will, after compilation, be executed on the host computer or a specially designated host node of the target system, as the *host process.* 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 Vienna Fortran program, contains the actual computation; input/output is represented by communication with the host process. 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 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.

Example 4.1: Splitting

```
! jacobi program excerpt before splitting
   PROGRAM JACOBI

   PROCESSORS R(2,2)
   REAL UHELP(N,N), U(N,N), F(N,N), RES(N,N) DIST(BLOCK,BLOCK)
           . . . . .
      READ (5,*) ITER, OMEGA
      IF (ITER .EQ. 0) STOP
              . . . . .
      DO I=1,ITER
         CALL JACOBI(OMEGA, F, U, UHELP, N)
         CALL RESID(I, RNORM, RES, F, U, N)
         RKONV = RNORM/RNORM0
         WRITE(6,*) I,' CONVERGENCE:', RKONV
         RNORM0 = RNORM
      ENDDO
   END

   SUBROUTINE NORM(ITER, RNORM, RES, N)
        . . . . .
     RNORM = SQRT(RNORM)/FLOAT(N-1)
     WRITE(6,*) ITER,'iterations: ',RNORM
   END
```

```
! host program after splitting:
      PROGRAM HOST
         . . . . .
      READ (5,*) ITER, OMEGA
      SEND ITER, OMEGA TO NODE
      IF (ITER .GT. 0) GOTO 2
          . . . . .
      RECEIVE ITER FROM NODE
      DO I=1,ITER
        CALL HOSTRESID
        RECEIVE I, RKONV FROM NODE
        WRITE(6,*) I,' CONVERGENCE:', RKONV
      ENDDO
 2    STOP
      END


      SUBROUTINE HOSTRESID()
        CALL HOSTNORM()
      END




      SUBROUTINE HOSTNORM()
      REAL RNORM
        RECEIVE ITER, RNORM FROM NODE
        WRITE(6,*) ITER,'iterations: ',RNORM
      END
```

```
! node program after splitting
      PROGRAM NODE
      PROCESSORS R(2,2)
        . . . . .
      RECEIVE ITER, OMEGA FROM HOST
      IF (ITER .EQ. 0) GOTO 2
         . . . . .
      SEND ITER TO HOST
      DO I=1,ITER
         CALL JACOBI(OMEGA, F, U, UHELP, N)
         CALL RESID(I, RNORM, RES, F, U, N)
         RKONV = RNORM/RNORM0
         SEND I, RKONV TO HOST
         RNORM0 = RNORM
      ENDDO
 2    STOP
      END

  SUBROUTINE RESID(ITER,RNORM,RES,F,U,N)
         . . . . .
      CALL NORM(ITER, RNORM, RES, N)
         . . . . .
  END

  SUBROUTINE NORM(ITER, RNORM, RES, N)
         . . . . .
      RNORM = SQRT(RNORM)/FLOAT(N-1)
      SEND ITER, RNORM TO HOST
   END
```

Example: Splitting

We use a (slightly modified) excerpt from the demonstration program in which a subroutine performing a Jacobi iteration is invoked to illustrate how this works. The sequential program as shown reads two input values; it then executes a loop *iter* times, which calls two subroutines, and writes a line of information. Neither of these subroutines performs I/O; however, subroutine RESID calls NORM, which writes two values.

The host program after splitting reads the input values and sends them to the node. Since the subsequent condition uses an input value, the value of the conditional does not need to be received from the node here, as is generally the case. If the compiler can be sure that the value of *iter* does not change before the DO loop is entered, it could eliminate the receive which follows and its corresponding send in the node program. The host loop invokes one of the subroutines: the host version has no arguments and performs no computation. It simply calls NORM to receive two values from the node subroutine and write them.

The node program contains all computation, but is still sequential.

4.4.2 Distribution Analysis

In this Section we describe the distribution analysis techniques as implemented in VFCS.

Distribution analysis is performed automatically by the system after the *Basic Parallelization* has been invoked. However, as explained below, there are some situations where the system does not have enough information to determine certain properties of the distribution of a dummy argument and thus the user will be prompted for further information.

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.

4.4.2.1 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, S_BLOCK, B_BLOCK*, 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.

Arrays with a standard distribution are further classified such that all arrays with identical shape and distribution are contained in the same *distribution class*.

4.4.2.2 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.

4.4.2.3 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.

Furthermore, a decision is made whether or not a dummy argument array will always be associated with a *standard distribution*. This can be done automatically, except for adjustable arrays, in which the user has to provide that information (see the example below). 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.

Another problem which is solved during distribution analysis arises from scalar dummy arguments which are bound at runtime to elements of non-replicated arrays. Such a dummy variable is transformed into an adjustable array with the same shape as the actual array. This may necessitate cloning if the arrays which elements are bound to the scalar variable in different call statements have different ranks.

4.4.2.4 Unclassified Dummy Arrays

Whenever the system can not determine whether a dummy array is a subarray of the corresponding actual array, the system presents the *Dummy Array Shape Window* to ask the user for that information. The system displays the actual array, the `CALL` statement, and the dummy array declaration. The following options are available to the user:

Yes	Yes, it is always a subarray.
Yes(call)	For all other pairs of arrays in this call, the condition is fulfilled.
Yes(all)	The condition is fulfilled in all other situations occurring during the execution of this transformation.
No	No, it may not be a subarray in at least one incarnation.

Example 4.2: Passing an array element to an adjustable array

```
REAL A(100,100,100) DIST(BLOCK,BLOCK,BLOCK)
       .......
DO I=1,100
  CALL F(A(1,1,i),N,M)
END DO

SUBROUTINE F(B,N,M)
REAL B(N,M) DIST(*)
      ........
```

In this example a scalar, i.e. the array element $A(1,1,i)$, is passed to a dummy argument which is an adjustable array. In such a situation the system can not determine whether the dummy array B is associated with a sub-array of A in all incarnations of F. Thus, the user is prompted for that information.

The *Dummy Array Shape Window* window is described in detail in Section C.3.3.1 of the Appendix.

4.4.2.5 Distribution Analysis for Workspaces

In the case of dummy arrays which represent a single virtual array in a workspace, the system does not have sufficient information to decide which virtual array is accessed at runtime. Furthermore, if more than one dummy array may represent a virtual array at runtime, the possible combinations of virtual arrays in incarnations have to be determined. In these cases the user has to assist the system as described below.

The Workspace Window

If a program uses workspaces which are distributed as described in Section 2.2.7, then the system will display the *Workspace Window* during the basic parallelization phase to prompt the user for further information. For each incarnation of a `CALL` statement with a workspace array (element) as actual argument the system asks to specify which virtual arrays are associated with the corresponding dummy arrays.

The system presents in the *Workspace Window* the following information:

- the `CALL` statement,
- the work spaces and their virtual arrays being accessed, and
- the last user-supplied answer to such a question (if this answer is syntactically possible in the current situation).

The user can select commands to determine all possible combinations of virtual arrays with dummy arrays in incarnations resulting from that call. The possible commands are:

Last	Determines the last answer to be the current answer.
New	Invalidates all combinations specified.
Show	Reproduces the information window.
End	Terminates the specification of combinations for this situation.
Exit	Stops the whole transformation process.
Save	Starts the protocol for all supplied combinations. The system requires a file name.
Read	If the information needed to perform distribution analysis for workspaces (i.e. the mapping of virtual arrays to dummy arrays) is contained in a file, this command causes the system to read appropriate commands from the file. The system requires a file name.

Besides these commands the user may specify the combinations of virtual arrays in the *Commandline*, using a special notation. A specification has to be made up of a single term for each dummy array accessing a virtual array of a workarray. A single term is either an integer constant **'n'** or a sequence **'n-m'** of constants. The specified constants have to be in the range between one and the number of virtual arrays in the corresponding workarray. Furthermore, if more than one sequence is given in a specification, the sequences must have the same number of elements. The combinations determined from such a specification are made up of the single constants and of the constants in the same positions of the sequences.

In Example 4.3 a code excerpt from a multigrid code is used to show the need for user assistance during distribution analysis for workspaces.

Example 4.3: Distribution Analysis for Workspaces

```
PROCESSORS R(4,4)

WORKSPACE MG:: G1(33,33) DIST(BLOCK,BLOCK),
               G2(17,17) DIST(BLOCK,BLOCK),
               G3(9,9)   DIST(BLOCK,BLOCK)

REAL U(40000) DIST WORKSPACE MG

DO K=1,2
   ...
   IDF=GSTART(K)
   NPF=NPOINTS(K)
   IDC=GSTART(K+1)
   NPC=NPOINTS(K+1)
   CALL RESTR (...,U(IDF),NPF,U(IDC),NPC,...)
   ...
END DO

SUBROUTINE RESTR (...,GF,NPF,GC,NPC...)
REAL GF(NPF,NPF),GC(NPC,NPC) DIST(*)
   ...
END
```

In this example three virtual arrays `G1,G2` *and* `G3` *are embedded into the workspace array* `U`*. When analyzing the call statement, the system is unable to determine which virtual arrays are accessed via the dummy arrays* `GF` *and* `GC`*. Thus, during distribution analysis the system will display the* Workspace Window *to ask the user which virtual arrays are associated with the dummy arrays* `GF` *and* `GC` *of subroutine* `RESTR`*. to a dummy argument which is an adjustable array. The user may enter in the command line of this window the following combinations*

```
1-2  2-3                        1  2
                  or            2  3
```

to specify that

- *in the first call to* `RESTR GF` *is associated with the first virtual array (i.e.* `G1`*) and* `GC` *is associated with the second virtual array (i.e.* `G2`*), and*
- *in the second call* `GF` *is associated with the second virtual array and* `GC` *with the third virtual array (i.e.* `G3`*).*

4.4.3 Masking

Masking is a transformation that 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 a `DO 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.`.

Example 4.4: Masking

```
! code before masking
PROCESSORS R(4)
REAL A(100),B(100) DIST(BLOCK)
      .......
   DO 10 I=2,98
      A(I) = B(I+2)
   END DO
```

```
! code after masking
PROCESSORS R(4)
REAL A(100),B(100) DIST(BLOCK)
      .......
   DO 10 I=2,98
      ....
      OWNED(A(I))-> A(I) = B(I+2)
   END DO
```

The mask `OWNED(A(I))` *ensures that on processor 2, for example, the assignment statement is executed only during the iterations 26 to 50.*

Moreover, communication statements that realize the host-node communication may be associated with a mask.

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.

VFCS currently uses different masks to distribute the work among the processors. The different masks are:

- `OWNED(ref)`:
  is used for masking of assignments to distributed arrays; `ref` is a reference to a distributed array;
- `MSP`:
  If a value of a replicated variable is output, only one of the node processors sends this value to the host process. This processor is determined automatically by VFCS and called *master of the segment processors*.
- `CSM(A)`:
  If a part of the distributed array `A` is allocated on more than one processor (partially replicated), than only one of these processors is responsible for communication. The *communication segment manager* is determined automatically by the system.
- `LS(A)`:
  Whenever elements of a distributed array `A` are input, this mask ensures that only processors that own these elements receive the input data form the host process.

4.4.4 Communication Insertion

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.

Example 4.5: Communication Insertion

```
! original code                      ! code after masking & communication insertion
PROCESSORS R(4)                      PROCESSORS R(4)
REAL A(100),B(100) DIST(BLOCK)       REAL A(100),B(100) DIST(BLOCK)
      .......                              .......
   DO 10 I=2,98                         DO 10 I=2,98
      A(I) = B(I+2)                        EXSR(B(I+2))
   END DO                                  OWNED(A(I))-> A(I) = B(I+2)
                                        END DO
```

The `EXSR` *statement ensures that private copies of non-local variables are updated if necessary. For example, on processor* 2 *in iteration* 49 *and* 50 *the array elements* $B(51)$ *and* $B(52)$ *are received from processor* 3.

Note that `EXSR` statements provide an abstraction of the actual communication. The semantics of `EXSR` statements is described in the Appendix B. During communication optimization, `EXSR` statements are extended by an *overlap description* that determines the storage requirements for the non-local variables. The relevant analysis is described in detail in Section 4.5. In the code generation phase `EXSR` statements will be replaced by corresponding calls to a message passing library.

4.5 Overlap Analysis

Overlap analysis is performed to determine which non-local elements of a distributed array[1] 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 may be used as:

- a measure for the amount of non-local data required
- a measure for the quality of the array's distribution
- a measure for the locality of certain computations

4.5.1 Overlap Area

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* as described next.

4.5.2 Overlap Description

An overlap description for an n-dimensional array is presented in the form

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

where dl_i and du_i denote the offsets with respect to the lower and upper bound of dimension i of the local segment.

Overlap descriptions are used by the system to significantly improve the organization of communication; it facilitates memory allocation and the local addressing of arrays. The relevant analysis is described in detail in [18].

The system displays overlap descriptions by appending them to the corresponding `EXSR` statements.

4.5.3 Computation of Overlaps

Overlap analysis is usually performed automatically by the system during the communication optimization phase. However, it may also be invoked by the user after basic parallelization by selecting the corresponding menu entry from the *Analysis* menu.

VFCS determines the overlap area for a distributed array in two steps as outlined below.

Step 1: Computation of an overlap descriptions for each *rhs* reference

Assume an assignment statement of the form $A(f_1, \ldots, f_n) = B(e_1, \ldots, e_m)$ which may appear in an arbitrarily nested loop, and where the scalar subscript expressions f_i, e_j are linear in the

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

loop variables.

If both A and B have the same shape and identical distributions then the offsets with respect to the lower and upper bound of each dimension (i.e. dl_i and du_i) are determined by a symbolic subtraction of the corresponding subscript expressions.

If the symbolic subtraction does not a yield a constant or the arrays have different shapes or distributions, then for each processor a system of inequalities, based on the bounds of the local segments of A and the subscript expressions f_i, is constructed. From this inequality system restrictions for the loop variables are determined and substituted for the subscript expressions of B to obtain a lower and upper bound for the references to B. These bounds are compared to the local segment of B yielding the overlap description with respect to a particular processor. The final overlap description for the array reference is then determined by computing the maxima and minima over all processors.

Step 2: Computation of overlap area by combining all overlap descriptions

The overlap area for a distributed array, say B, is obtained by means of interprocedural overlap analysis and thus takes the overlap descriptions of all references to B into account.
The overlap area for distributed arrays is displayed by the system in a separate window after selecting the *Overlap* menu entry from the *Analysis* menu.

In Example 4.6 the Jacobi code after basic parallelization and after overlap analysis is shown and the overlap area of array U for a particular processor is visualized.

Note that during overlap analysis unnecessary `EXSR` statements are eliminated.

Remark:

There is one possible error message:

Process without local segment of array ...

that indicates that a processor may access a non-local element of an array of which it does not have an own local segment. Such non-local elements cannot be described by the overlap concept and thus cannot be updated correctly. If this message appears the user is forced to change the distribution of the array.

4.5.4 Total Overlap

Whenever there is insufficient static information to determine the value of a component in the overlap description, a conservative estimate has to be made. This may mean that the remainder of an array dimension is included in the overlap area. This is called a *total overlap* (in a particular dimension).

A total overlap in a particular array dimension is represented by means of angle brackets in the overlap description, e.g.

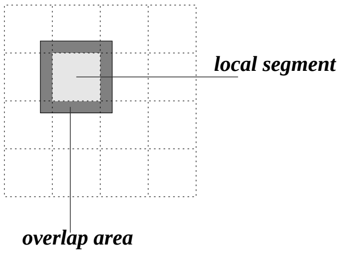
$$[< / >, < / >]$$

For example, the overlap concept is not an appropriate approach if there is any form of indirection in the code. In such a case, alternative methods must be employed. For a further discussion of this, see Section 4.7.2.

Example 4.6: Overlap analysis

```
! code after basic parallelization
  PROCESSORS R(4,4)
  REAL F(N,N), U(N,N), UHELP(N,N) DIST(BLOCK,BLOCK)
      .......
  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(I2D,J1D)
      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
  PROCESSORS R(4,4)
  REAL F(N,N), U(N,N), UHELP(N,N) DIST(BLOCK,BLOCK)
      .......
  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
```



This example shows the jacobi kernel obtained after overlap analysis. The overlap descriptions are appended to the corresponding EXSR *statements. In the figure, the overlap area and the local segment for array* U *on a particular processor is visualized.*

reasons for a "total overlap" and how to avoid it

4.6 Optimizing Transformations

Optimizing transformations fall into two categories:

- Optimization of communication
- Optimization of masking

4.6.1 Optimization of Communication

This transformation extracts communication from `DO`-loops wherever possible and thus combines single element messages to larger ones. It may be activated by the user by selecting the *Communication Optimization* from the *Parallelization* menu.

During this transformation interprocedural overlap analysis as described in the previous section is performed and each `EXSR` statement is associated with an overlap description.

Example 4.7: Communication Optimization

```
! code before communication optimization
PROCESSORS R(4)
REAL A(100),B(100) DIST(BLOCK)
       .......
   DO 10 I=2,98
     EXSR(B(I+2))
     OWNED(A(I))-> A(I) = B(I+2)
   END DO
```

```
! code after communication optimization
PROCESSORS R(4)
REAL A(100),B(100) DIST(BLOCK)
       .......
   EXSR B(:) {[0/2]}
   DO I1D=2,98
     OWNED(A(I1D))-> A(I1D) = B(2 + I1D)
   ENDDO
```

In this example the `EXSR` *statement is extracted from the loop and thus many single element messages are substituted by a single large message. The array section* `B(:)` *indicates that the* `EXSR` *statement will result in the communication of an array section rather than an array element.*

The applicability of this transformation depends on the dependence graph of the loop. If a loop contains loop carried true dependencies then extracting the communication statements from the loop is usually not possible.

4.6.2 Optimization of Work Distribution

After basic parallelization 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. In the following transformations that eliminate the overhead introduced by the evaluation of statement mask are described.

4.6.2.1 Optimization of Masking

The following transformations optimize the handling of masks in loops and, in many cases, lead to the *strip-mining* of the loop across the processors by partitioning the iteration space:

1. *Iteration Elimination*: Iteration elimination deletes irrelevant statement instances by eliminating an entire loop iteration for a process. This transformation will adapt `DO` loops wherever possible so that each processor executes its own set of loop iterations. This is done by enforcing the masks by an appropriate adaptation of the loop bounds in the internal representation. VFCS displays an adaptation of loop bounds by prefixing the loop header with "`mask->`" as shown in the example below.
2. *Mask Simplification*: If it can be shown that a mask is always *true* for each instance of each process, it is eliminated.

Iteration elimination and mask simplification are performed automatically by the system after the *Mask Optimization* entry from the *Parallelization* menu has been selected.

Example 4.8: Mask Optimization

! code before mask optimization	! code after mask optimization
PROCESSORS R(4)	PROCESSORS R(4)
REAL A(100),B(100) DIST(BLOCK)	REAL A(100),B(100) DIST(BLOCK)
....	....
EXSR B(:) {[0/2]}	EXSR B(:) {[0/2]}
DO I1D=2,98	mask-> DO I1D=2,98
OWNED(A(I1D))-> A(I1D) = B(2 + I1D)	A(I1D) = B(2 + I1D)
ENDDO	ENDDO

This example shows the effect of mask optimization. The mask of the assignment statement (i.e. `OWNED(A(I1D))->`*) is eliminated and the loop header is associated with a mask. In the code obtained after code generation the loop bounds are automatically adapted by the system such that for example processor 1 executes only iterations 2 to 25.*

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 either be transformed (i.e. by applying scalar forward substitution or induction variable substitution) prior to the optimizations steps or mask propagation as described next should be performed.

4.6.2.2 Mask Propagation

Mask propagation is a transformation that propagates the mask of statements to other statements in the same loop nest. Thus it may eliminate the execution of assignments to scalar variables if the computed values are not needed by the executing process. It may be selected from the *Parallelization* menu.

In contrast to the other steps outlined here, the selection of *Mask Propagation* is not essential – it is not an error to omit this transformation. It is advantageous to use this transformation before the two optimization transformations, in particular when there are assignments to scalar variables within `DO`-loops in a unit. Note that in contrast to the other transformations, this is performed within the active unit only.

Example 4.9: Mask Propagation

! code after basic parallelization	! code after mask propagation
PROCESSORS R(4)	PROCESSORS R(4)
REAL A(100),B(100) DIST(BLOCK)	REAL A(100),B(100) DIST(BLOCK)
....	....
DO I=2,99	DO I=2,99
EXSR A(I+1)	EXSR A(I+1)
EXSR B(I-1)	EXSR B(I-1)
S=B(I-1)+A(I+1)	OWNED(A(I))-> S=B(I-1)+A(I+1)
OWNED(A(I))-> A(I)=2*S	OWNED(A(I))-> A(I)=2*S
END DO	END DO

This example shows the effect of mask propagation. The mask of the assignment statement to `A(I)` *is propagated to the assignment statement to S. As a consequence, adaptation of the loop bounds during mask optimization can be applied successfully.*

4.7 Transformation of Parallel Loops

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

4.7.1 Work Distribution for DO INDEPENDENT Loops

Whereas the work distribution is usually derived automatically from the data distribution, the distribution of loop iterations of `DO INDEPENDENT` loops may be specified explicitly by the user.

The work distribution for `DO INDEPENDENT` loops (i.e. the mapping of loop iterations to processors) may be interactively modified by the user via the *Work Distribution* entry in the *Parallelization* menu. **Note:** This may not be selected before the *Basic Parallelization* transformation has been performed. Four interactive options are provided for work distribution; the user may also reselect the system-determined distribution. The options enable loop iterations to be distributed

- to specific processors,
- to the owner of an array element,
- automatically (i.e. by the compiler)
- or in a block
- or cyclic fashion.

If the user does not specify the distribution of loop iterations, the system chooses a default work distribution.

Note that the user may interactively transform a `DO` loop into a `DO INDEPENDENT` loop (before the *basic parallelization* by selecting the corresponding menu entry form the *Transformations* menu.

`DO INDEPENDENT` loops whose iterations are not independent are illegal and the results from executing such programs are undefined. Thus, if the system cannot prove during the *Basic Parallelization* phase that the iterations are independent, or scalar variables are used within the loop before receiving a new value there, the user is warned of this and asked to confirm the transformation to a `DO INDEPENDENT` loop. Furthermore, the user is provided with information on the use of arrays and variables within this loop.

Restrictions on loops which may be transformed into `DO INDEPENDENT` loops, are described in the Appendix.

4.7.2 Parallelization of DO INDEPENDENT Loops

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

VFCS automatically parallelizes `DO INDEPENDENT` loops after the *Code Generation* entry from the parallelization menu has been selected. Note that prior to the code generation the user may modify the work distribution of `DO INDEPENDENT` loops by selecting *Work Distribution* entry from the *Parallelization* menu.

This process combines compile time parallelization techniques with run-time analysis. Information from dataflow analysis, data dependence analysis, data distribution and overlap analysis is utilized. To implement runtime analysis and parallel execution, the (slightly modified) set of Parti primitives developed at ICASE NASA is used. The parallelization techniques utilize the fact that (due to the restrictions on such loops) 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. 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. Thus, 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.

VFCS integrates the parallelization of `DO INDEPENDENT` loops and conventional `DO` loops within a single program, so that if an array is accessed both regularly and by an index array, a `DO INDEPENDENT` loop may be used for the latter portions of the code and this will not affect compilation of the rest of the program: storage space is reserved for it according to overlap analysis in the usual manner. However, the system must also reserve space for non-local data read and written during the execution of `DO INDEPENDENT` loops and to do so it must make certain assumptions about the amount of memory required for local storage of such data.

Transformation of `DO INDEPENDENT` Loops

Each `DO INDEPENDENT` loop is transformed into three main phases: the *work distributor*, the *inspector*, and the *executor*. On each processor, the work distributor calculates the *execution set*, i.e. the set of loop iterations to be executed on that processor. 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 so called *schedules*, that control runtime procedures moving data among processors in the subsequent executor phase, and establishes 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 are produced from the appropriate *global reference* lists, along with the knowledge of the array distributions.

If there is any indirectly distributed array in the program *constructor of data distribution descriptors* code is generated which creates the runtime distribution descriptor (RDD). The RDD is represented by two data structures: the *local index set* and the *translation table*. The local index set of an array in each processor is an ordered set of indices designating those elements of the array that have to be allocated on that particular processor. The translation table is a distributed data structure which records the home processor and the local address in the home processor's memory for each array element.

Global reference lists are constructed from the subscript functions of the accessed arrays. Results of global to local index conversion are contained in the corresponding *local reference* lists. The executor performs communication described by schedules (gather/scatter non-local array elements used/updated in the loop), and executes the loop body for 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.

For a general discussion of the *inspector-executor paradigm* see [25].

4.8 Target Code Generation

The final step in parallelizing an application is target code generation which is invoked by selecting the menu item *Code Generation* from the *Parallelization* menu. This results in the generation of an explicitly parallel message passing Fortran 77 program.

Target code generation includes the following actions:

- Adaptation of declarations and references to distributed arrays according to the data distribution;
- Transformation of high level communication descriptors (EXSR) into calls to target architecture and environment specific communication procedures;
- Substitution of *masks* by transforming masked assignment statements into conditional statements and adapting loop bounds.

4.8.1 Target Architectures and Communication Libraries

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.

4.8.1.1 Code Generation Options

The following options may be selected from the *Code Generation - Setup* window. The user can select both target architecture and communication library by clicking on radio buttons. The switches are selected by checking the appropriate checkboxes.

1. Selection of target architecture
   - Intel iPSC/860
   - Intel Paragon
   - Meiko CS-2
2. Selection of communication library
   - Intel NX
   - Parmacs 5.1
   - Parmacs 6.0
   - MPI
3. Selection of communication model
   - Host/Node Model
   - Node Only
4. Special Options
   - Byte Swap
     If the *byte swap* switch is set, VFCS generates a call to particular byte swap routines after each host-node communication. This is required because some architectures use different byte orderings for the host and the node computers (eg. node = iPSC 860, host = PC 386 like in some hypercube configurations).

- Communication Protocol
  If the *communication protocol* switch is set, all communications in the program will be protocoled.

All parameters can be set multiple times during a VFCS session. It is also possible to select the current defaults by clicking on the 'default' pushbutton. When VFCS is left, the current parameter set becomes the default.

4.8.1.2 Saving the Target Code

The generated parallel program is saved to file by selecting the menu entry *Save Target Code* from the *File* menu. Usually the parallel program is saved in several files according to the conventions described in Section 3.3.2

Setting an Output File Name Prefix

Selecting on the *Select Project Name* menu entry from *File* menu opens the **Select Project Name** dialog. The user can select a project name which is used as prefix for the generated output files. This approach makes it easy to identify files belonging to a 'project', for example if they are to be transferred to the target architecture or simply put into an archive.

4.9 Utilities

4.9.1 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.

Version management is invoked by selecting the corresponding entry from the *File* menu. The following functions are provided:

Set Version: The current state of the parallel program is saved to file under the name given by the user.

Get Version: Restarts the parallelization process at the specified version.

Delete Version: Old versions which are not further needed can be deleted via this menu item.

Delete All Versions: All program versions previously saved are deleted.

4.9.2 The Protocol Facility

The *protocol facility* enables the user to record the actions of an interactive VFCS session in a *protocol file*. In a protocol file the actions of a parallelization session are specified by means of driver directives. Therefore protocol files of a complete parallelization session may be used as driver files for running VFCS in interactive mode as described in Chapter 3.

4.9.2.1 Automatic Generation of Driver Files

If during an interactive VFCS session the protocol mode is switched on, all actions taken by the user are recorded and may be saved to a protocol file which may be used as a driver file in automatic mode.

Note: In the current version of VFCS the user should modify automatically generated driver files by adding the command `const_prop` after the `load_sources` command. This is necessary because currently FCS performs constant propagation automatically only in interactive mode.

Turning the Protocol Mode On

The protocol facility is invoked by selecting the *Protocol On* entry from the the *Options* menu. The system then displays a file selector box where the name of the protocol file may be specified.

Turning the Protocol Mode Off

The protocol facility can be turned off by selecting the *Protocol Off* entry from the the *Options* menu. In this case, the actual protocol file is saved and closed.

4.9.2.2 Manual Generation of Driver Files

Driver files may be created manually using a text editor. The syntax of the driver directives is described in Appendix D.

The selection of program constructs is not supported if you generate driver files manually. Transformations which require a selection of program constructs should only be used in automatically generated driver files (see Appendix D for details).

Chapter 5

Target Compilation and Execution

5.1 Generating an Executable Program

5.1.1 Generated Output by VFCS

VFCS generates a buildfile, a makefile, two message passing Fortran 77 source files, and several data files containing runtime information. The names of the files are:

> vfbuild.*project*, Make.*project*, *project*_host.f, *project*_node.f, *project*_init.nb, *project_init.nsb*, *project_init.ob*, *project*_init.ovp, *project*_init.sb, *project*_init.tov, *project*_init.ws,

where *project* is the project name selected via the *Select Project Name* dialog if VFCS has been executed in interactive mode, or with the *set_project* command or the *-p* option if VFCS has been executed in automatic mode. If no project name is selected, the files get default filenames (vfbuild, Makefile, host.f, node.f, ...).

5.1.2 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 which are located in the *$VFCSROOT/lib/make* directory[1].

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 linking steps

To generate an executable parallel program, the following steps have to be performed:

- copy the output of VFCS (i.e. the 11 above mentioned files) to the target machine
- start the build process with

  ```
  source vfbuild.project
  ```

  (or simply `source vfbuild` in case that no project name has been selected[2]).

[1] the environment variable *$VFCSROOT* is described in the installation section.

[2] The VFCS 1.2 distribution contains buildfile templates for a number of different shells (sh, csh, tcsh, ksh, bash). Please refer to the installation section.

VFCS currently supports two different execution models, the *host-node model* and the *node-only model* which can be selected by the user in the *Code Generation* options of the *Options* menu. In the first case two executables are generated, whereas in the second case only one executable is generated.

5.1.2.1 Host-Node Execution Model

If the host-node execution model has been chosen the build process generates two executables

- *project*_host
- *project*_node

containing a host program and a node program, respectively.

5.1.2.2 Node-Only Execution Model

If the node-only execution model has been chosen the build process generates a single executable

- *project*_node

containing the node program.

5.2 Executing the Parallel Application

The procedure to run the parallel program is dependent on the execution model. In the following we assume that our project name has been set to *project*.

Host-Node Model

If a hosted model is chosen, it is sufficient to call the host program by name, i.e. simply type

project_host

on the command line.

Node-Only Model

In a pure SPMD 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.

5.2.1 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). Please do not change them by hand.

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).

Some systems require additional resource management, e.g. the selection of an appropriate partition. This cannot be done automatically and therefore has to be done by the user. Here are some hints:

Intel iPSC 860

On the Intel iPSC 860 currently only the host-node model is available. To run the parallel application, the host program has to be executed. Cube allocation, loading of the node program and cube deallocation are done automatically by the host program.

Meiko CS-2

Before starting a program an appropriate partition has to be chosen. This is done by setting the *RMS_PARTITION* environment variable. The availability of partitions can be checked with the *rinfo* command (see the Meiko CS-2 manual for a more detailed explanation).
If an appropriate partition is selected it is sufficient to start the program either by typing its name on the command line (hosted model) or via the *prun* command. Sometimes it is useful to set the *RMS_VERBOSE* environment variable, this allows the user to see more details about what is happening in the system.

5.2.2 Runtime Environment for Performance Measurings with VFPMS

At each invocation of the measuring routine *TIMEAS*[3] a timestamp 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.[4] 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[5] 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.[6]

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

[3] a call to a *TIMEAS* corresponds to a measuring point calculated automatically by VFPMS (see section 7.4)
[4] i.e. the buffer can contain 10000 timestamps.
[5] 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.
[6] this corresponds to the $100ns$ resolution of *dclock()* on the iPSC.

Chapter 6

VFCS Analysis and Transformation Tools

In this chapter, we describe the *analysis services* and the *program transformations* provided by VFCS. The presentation here is rather short and makes extensive use of examples. For a more complete discussion of program analysis and basic transformations, with definitions of the major concepts and the precise specification of the applicability and the effect of transformations see *Zima and Chapman: Supercompilers for Parallel and Vector Computers (ACM Press – Addison-Wesley, 1991).*

6.1 Program Analysis

The integrated *program analysis tool* of VFCS computes and visualizes analysis information.

The program analysis performed by VFCS includes:

- Data Flow Analysis
- Dependence Analysis
- Interprocedural Analysis
- Overlap Analysis

The entirety of information sets computed for a program is referred to as the *program information set.*

6.1.1 Dataflow Analysis

Data flow analysis determines *data flow chains*, *I/O* lists, and the *control flow graph.*

6.1.1.1 Data Flow Chains

The data flow analysis service computes the following information sets for all occurrences of scalar variables:

1. For every definition of a variable both the definition-use chains and the output dependences.
2. For every usage of a variable both the use-definition chains and the anti dependences.

To obtain these information sets the user has to select *Dataflow* from the *Analysis* menu. This will result in both the computation of the information mentioned above and the creation of a window. The user is then prompted to select a line which contains an occurrence of the variable(s) to be analyzed.

The following options exist:

1. *DU chains*
   This will result in showing the definition-use chains for all scalar variables defined in the selected statement.
2. *Output chains*
   This will result in showing the output dependences for all scalar variables defined in the selected statement.
3. *Anti chains*
   This will result in showing the anti dependences for all scalar variables used in the selected statement.
4. *UD chains*
   This will result in showing the use-definition chains for all scalar variables which are used in the selected statement.
5. *All chains*
   This will result in showing all the chains mentioned above for all scalar variables occurring in the selected statement.

Every chain is divided into lines of the form:

$$var : (row\_nr1 - -- > row\_nr2)chain\_type/def\_type/dep\_type$$

where

- *var* identifies the name of a scalar variable
- *row_nr1* identifies the line selected
- depending on the kind of chain
  - *row_nr2* identifies a line where the value of VAR,
    1. as defined in line *row_nr1*, is used.
    2. as defined in line *row_nr1*, is changed.
    3. as used in line *row_nr1*, is changed.
    4. as used in line *row_nr1*, was defined.
  - *chain_type* identifies the kind of chain
    1. DU stands for definition-use chain
    2. OP stands for output chain
    3. AT stands for anti chain
    4. UD stands for use-definition chain
  - *def_type* is set only for scalar variables defined and identifies the kind of definition (either 'must' or 'may')
  - *dep_type* is set only for dependence information of the first three categories and identifies the kind of dependence (either 'car' for loop carried dependence or 'ind' for loop independent dependence)

6.1.1.2 Use/Def Sets

Selecting the menu entry *Use/Def Sets* from the *Analysis* menu will display the variables written (marked as 'MAY_DEF' or 'MUST_DEF') or read (marked as 'USE') for every statement (identified by row number and statement type) for the actual program unit.

6.1.1.3 Flowgraph

Selecting the menu-item *Flowgraph* from the *Analysis* menu will display the control flow graph. For each statement (represented by the row number and the statement type) both all successors (related to control flow) and all predecessors, which are identified by row numbers, are shown.

6.1.2 Dependence Analysis

The dependence analysis service computes the following information sets for all occurrences of all variables within some `DO`-loop nest:

1. For every definition of a variable both the true dependences and the output dependences
2. For every usage of a variable the anti dependences

To obtain these information sets the user has to select *Data Dependences* from the *Analysis* menu.
The system then displays a submenu which enables the user to restrict the to be displayed as follows:

1. Selection of the dependence type:
   - *All dependences*
     This will result in selecting all dependences for display for all variables occurring in the selected region.
   - *True dependence*
     This will result in selecting only the true dependences for display for all variables defined in the selected region.
   - *Anti dependence*
     This will result in selecting only the anti dependences for display for all variables used in the selected region.
   - *Output dependence*
     This will result in selecting only the output dependences for display for all variables defined in the selected region.
2. Selection of a *region*: A *region* may either by a `DO` loop or a single statement.
   - If the user clicks at a do loop header the system will display all the dependences (of the selected type and level) between all the statements in the body of that loop.
   - If the user selects a statement (other than a loop header) then the system will display all the dependences (of the selected type and level) that statement is involved in.
3. Selection of dependence level:
   - loop carried dependencies with a level between 1 and 9
   - loop independent dependencies (i.e. with level 0)
   - all dependences (loop carried and loop independent)

The dependence information is visualized in a separate window as a *dependence graph*. Each edge of the dependence graph is represented in the following form:

row_nr1 —> (level1 [:level2], dep_type, var1, [var2]) —>row_nr2

where

- *row_nr1*
  identifies the statement (by displaying the row number) which is the source of the dependence
- *level1*
  specifies the dependence level of the displayed dependence
- *level2*
  can be shown only for scalar variables; if specified, *level1 : level2* gives the range of levels for which the displayed dependence exists
- *dep_type*
  specifies the dependence type of the displayed dependence
- *var1*
  denotes the variable of the statement in line *row_nr1* which causes the dependence
- *var2*
  denotes the variable of the statement in line em row_nr2 which causes the dependence; it is omitted if *var2* is the same as *var1*
- *row_nr2*
  identifies the statement (by displaying the row number) which is the sink of the dependence

6.1.3 Call Graph

Selecting the menu entry *Call Graph* from the *Analysis* menu will display the call graph of the (host and node) program. The call graph is the basis for interprocedural analysis.

The call graph is displayed in a separate window. Each node represents a procedure of the program and each edge from a node A to a node B indicates that procedure A may call procedure B.

Double clicking at a node of the call graph has the effect that the corresponding unit will become the actual unit and is displayed in the *Program Window* of VFCS.

6.1.4 Procedure Calls

Selecting the menu item *Common Blocks* from the *Analysis* menu will display the *call vectors* of the actual unit. Each call vector is associated with a single incarnation of the actual program unit; it is defined as the vector of actual argument distributions that are associated with that particular incarnation.

6.1.5 Common Blocks

Selecting the menu item *Common Blocks* from the *Analysis* menu will display the dummy arguments and the variables of `COMMON` blocks accessed by the actual program unit. Every entry is marked as 'use_def' , 'may_def' or 'must_def', dependent on the first access.

6.1.6 Data Distribution

Selecting the menu item *Data Distribution* from the *Analysis* menu will display all distribution specifications of the entire program in a separate window.

6.1.7 Overlap Analysis

Each local processor segment associated with an array is extended by a contiguous rectilinear *overlap area* surrounding the segment which provides buffer space for copies of non-local variables accessed in the processor (see Section 4.5).

Overlap analysis is performed for each reference to a distributed array in the actual unit after selection of the menu entry *Overlap* of the *Analysis* menu. Overlap analysis may be restricted by the user to the actual program unit or it may be performed for the whole program.

The system represents overlaps in the *Program Window* by means of *overlap descriptions*.

Overlap Description

An overlap description for an n-dimensional array is presented in the form

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

where dl_i and du_i denote the offsets with respect to the lower and upper bound of dimension i of the local segment.

Whenever there is insufficient static information to determine the value of a component in the overlap description, a conservative estimate has to be made. This may mean that the remainder of an array dimension is included in the overlap area. This is called a *total overlap* (in a particular dimension).

A total overlap in a particular array dimension is represented by means of angle brackets in the overlap description, e.g.

$$[< / >, < / >]$$

The system displays overlap descriptions by appending them to the corresponding `EXSR` statements. Furthermore, the user may look at all overlap descriptions for the actual program unit or for the whole program.

For more information on overlap analysis refer to Section 4.5.

6.2 Program Transformations

VFCS provides three classes of program transformations:

- *Normalization transformations*
  are performed automatically by the system whenever a program is loaded in order to simplify subsequent compiler operations. Normalization transformations are described in Section 6.2.1.

- *Standard transformations*
  may be applied selectively by the user in order to adapt the program in such a way that subsequent parallelization transformations yield better results. Many of theses transformations are also useful for performance tuning of sequential programs. Standard transformations are described in Section 6.2.2.

- *Parallelization transformations*
  i.e. splitting, masking communication insertion and several optimizing transformations, are needed to transform a Vienna Fortran or HPF program into a parallel message passing program. Parallelization transformations are described in Section 4.4 - Section 4.7.

6.2.1 Normalization Transformations

Program normalization has the purpose of generating a standard form of the program in order to simplify subsequent compiler operations.

Whenever a program is loaded VFCS automatically performs the following normalization transformations:

- `DO` loop normalization.
- If conversion.
- Transformation of `IF` statements.
- Transformation of `GO TO` statements.
- Transformation of expressions.
- Substitution of statement functions.
- Transformation of `EQUIVALENCE` statements.
- Moving all `FORMAT` and `DATA` statements to the end of the declaration part.
- Elimination of unreachable or useless statements.
- Elimination of useless labels.

In the following, some of these transformations are illustrated by examples.

6.2.1.1 Normalization of DO Loops

- DO loops are normalized by introducing a new loop variable and enforcing a lower bound of 1 and step size 1. Each occurrence of the original loop variable is replaced by an appropriate linear function of the new loop variable.

- Each DO loop is terminated by a loop-specific CONTINUE statement.

- RETURN statements inside a do loop are substituted by a conditional GOTO statement.

Example 6.1: Normalization of DO Loops

```
! original code
      DO 10 I=2,100,2
         DO 20 J=1,10
            C(I,J) = V
20       CONTINUE
         DO 10 J=1,20
10          D(I,J) = V
```

```
! transformed code
      DO 11 I1D=1,50
        DO 20 J2D=1,10
          C(2 * I1D,J2D) = V
20      CONTINUE
        DO 12 J2D00=1,20
          D(2 * I1D,J2D00) = V
12      CONTINUE
11   CONTINUE
```

Example 6.2: Transformation of RETURN statements

```
! original code
   SUBROUTINE SUB(X)
   REAL X(100)

   DO 10 I=1,100
      X(I) = ...
      IF (X(I+1).GT.0) RETURN
   END DO
   ...

   RETURN
   END
```

```
! transformed code
      SUBROUTINE SUB(X)
      LOGICAL EX1
      INTEGER I1D
      REAL X(100)

      EX1 = .TRUE.
      DO 10 I1D=1,100
        IF (EX1) X(I1D) = ...
        IF (EX1) EX1 = .NOT.X(1 + I1D) .GT. 0
      END DO
      IF (.NOT.EX1) GOTO 1
   ...
1     END
```

RETURN *statements inside a do loop are substituted by a conditional* GOTO *statement.* RETURN *statements occurring at the end of procedures will be eliminated.*

6.2.1.2 If Conversion

If conversion is a transformation that changes all control dependences of a loop caused by GO TO statements and IF statements into data dependences in order to simplify data dependence analysis. During if conversion for each assignment statement the precise condition for its execution is computed and all assignment statements are transformed into corresponding conditional statements.

Example 6.3: IF Conversion

<pre>! original code
      DO I=1,100
         A(I)= ...
         IF (A(I) .GT. 0) THEN
            A(I+1) = ...
         ELSE
            A(I-1) = ...
         END IF
      END DO</pre>
	<pre>! transformed code
	    DO I1D=1,100
	       A(I1D) = ...
	       BR1 = .NOT.A(I1D) .GT. 0
	       IF (.NOT.BR1) A(1 + I1D) = ...
	       IF (BR1) A((-1) + I1D) = ...
	    ENDDO</pre>

6.2.1.3 Transformation of IF statements

- An arithmetic IF statement is transformed into conditional GOTO statements.
- Block-if statements are substituted by logical-if statements.

Furthermore, inside DO loops if conversion is performed as described in 6.2.1.2.

Example 6.4: Transformation of Arithmetic IF

<pre>! original code
 	
       if (i) 10,20,30
 	
10   ...
20   ...
30   ...</pre>
	<pre>! transformed code
	      AIF1 = I
	      IF (AIF1 .LT. 0) GOTO 10
	      IF (AIF1 .EQ. 0) GOTO 20
	      IF (AIF1 .GT. 0) GOTO 30
	10 ...
	20 ...
	30 ...</pre>

Example 6.5: Transformation of Block-if

<pre>! original code
      IF (I.GT.0) THEN
         CALL SUB1(A)
      ELSE
         CALL SUB2(A)
      ENDIF
      WRITE(*,*) A,B,C</pre>
	<pre>! transformed code
	      IF (.NOT.I .GT. 0) GOTO 3
	      CALL SUB1(A)
	      GOTO 2
	3 CALL SUB2(A)
	2 WRITE(*,*) A,B,C</pre>

6.2.1.4 Transformation of GOTO statements

- Computed GOTO and assigned GOTO statements are substituted by a conditional GOTO statements.

Example 6.6: Transformation of Computed GOTO

! original code	! transformed code
	`CG1 = A`
`goto (10,20,30) a`	`IF (CG1 .EQ. 1) GOTO 10`
	`IF (CG1 .EQ. 2) GOTO 20`
`10 ...`	`IF (CG1 .EQ. 3) GOTO 30`
`20 ...`	`10 ...`
`30 ...`	`20 ...`
	`30 ...`

6.2.1.5 Normalization of Expressions

- Standardization of Subscript Expressions within Loops

  Let $I_1, \ldots, I_n$ denote the loop variables of a nested DO loop of depth n, and let e be a linear subscript expression in $I_1, \ldots, I_n$. Then e is transformed into $a_0 + a_1 * I_1 + \ldots + a_n * I_n$, where none of the a_i depends on a loop variable.

- Handling of Expressions With Side Effects

  If a statement contains an expression with a reference to a function which may produce side effects, then the function reference is substituted by a new scalar variable and an appropriate assignment of the function to that scalar is inserted prior to the statement. Note, that this transformation is performed during initial parallelization.

- Expression Simplification

  Expressions are simplified wherever possible. The expression simplification may also be selected manually from the Transformations menu.

- Substitution of Statement Functions

  Each reference of a statement function is textually replaced by the definition of the statement function and formal parameters are replaced by the corresponding actual parameters.

Example 6.7: Standardization of Subscript Expressions

```
! original code
DO I=1,10
  DO J=1,10
    DO K=1,10
      A(I) = A(4*K+3*J+2*I+I-99+J)
    END DO
  END DO
END DO
```

```
! transformed code
DO I1D=1,10
  DO J2D=1,10
    DO K3D=1,10
      A(I1D) = A((-99) + 3*I1D + 4*J2D + 4*K3D)
    END DO
  END DO
END DO
```

Example 6.8: In-line Expansion of Statement Functions

<table><tr><td>

```
! original code
   program stmtfun
   real x,y,a(10)

   stf1(x) = x*x+3*x

   read(*,*) y

   do 10 i=1,10
      a(i) = stf1(y)
   enddo

   write(*,*) a

   end
```

</td><td>

```
! transformed code
   PROGRAM MAIN

   INTEGER I1D
   REAL ARGEXP,STF100
   REAL X,Y,A(10)

   READ (*,*)Y
   DO 10 I1D=1,10
      ARGEXP = Y
      STF100 = ARGEXP * ARGEXP + 3 * ARGEXP
      A(I1D) = STF100
   END DO
   I = I1D
   WRITE(*,*) A
   END
```

</td></tr></table>

6.2.1.6 Transformation of `EQUIVALENCE` Statements

`EQUIVALENCE` statements are transformed such that only scalar variables or only array variables share the same storage. Furthermore, different `EQUIVALENCE` statements (referencing arrays only) are combined.

Example 6.9: `EQUIVALENCE` Transformation

<table><tr><td>

```
! original code
  program
  PARAMETER (N=10)
  integer a(N)
  real b(5),c(5),d(5)

  EQUIVALENCE (A,B),(B,C),(B,A)
      . . . . .
  EQUIVALENCE (C,A),(C,D),(A,D)

  read(*,*) a
  do 10 i=1,N
     a(i) = 0
  end do
  ....
```

</td><td>

```
! transformed code
  PROGRAM MAIN

  INTEGER I1D
  INTEGER A(N)
  REAL B(5),C(5),D(5)
  EQUIVALENCE (A,B),(B,C),(B,A),(C,A),(C,D),(A,D)

  READ (*,*)A
  DO 10 I1D=1,10
     A(I1D) = 0
  END DO
  .....
```

</td></tr></table>

6.2.2 Standard Transformations

VFCS provides the following standard transformations:

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

All the standard transformations can be selected from the *Transformations* menu.
Any component of the program information set that is required to test the applicability of a transformation must be present at the time a transformation is initiated. If necessary, the system will automatically update the information set at that time. The applicability of transformations is automatically checked by the system.

After completion of a transformation, the program information set will in general be updated, and dead code will be eliminated. If nothing else is said, transformations will be applied to the actual unit on an intraprocedural basis.

6.2.2.1 Constant Propagation

Constant propagation can be formulated as the problem of determining the value of a variable at a given point of the program whenever this value is constant. Constant propagation replaces all expressions which can be evaluated to a compile-time constant by that value. This is done for the whole program, i.e. constant propagation is performed on an *interprocedural basis.*

Example 6.10: Constant Propagation

```
! original code                          ! transformed code
      P=11                                     P=11
      DO 10 I=1,10                             DO 10 I=1,10
         DO 20 J=1,10                             DO 20 J=1,10
            B(I,J)=B(I,P)+1                          B(I,J)=B(I,11)+1
20       CONTINUE                        20       CONTINUE
10    CONTINUE                           10    CONTINUE
```

6.2.2.2 Scalar Forward Substitution

Programmers often reduce the number of redundant computations by using a scalar variable as a temporary variable to store the value of a common subexpression. However, subscript expressions containing such a variable enforce worst-case assumptions about dependences, which can result in a significant deterioration of the parallelization process. *Scalar forward substitution* replaces – under appropriate conditions – an applied occurrence of such a variable by the expression on the right-hand side of an assignment to that variable.

Example 6.11: Scalar Forward Substitution

```
! original code                          ! transformed code
      DO 10 I=1,10                             DO 10 I=1,10
         K=2*I                                    A(2*I+1)=A(2*I)+5
         A(K+1)=A(K)+5                   10    CONTINUE
10    CONTINUE
```

6.2.2.3 Induction Variable Substitution

Induction variable substitution deals with a kind of variable which is sometimes introduced to improve program performance on a sequential computer, but which can seriously weaken the precision of dependence tests. Induction variables are scalar integer variables which are incremented or decremented by a constant amount in each iteration of a do-loop and are mostly employed to get around the restrictions imposed upon permissible subscript expression in early Fortran standards. Induction variable substitution replaces all uses of induction variables in a loop of the actual unit with a linear function of the loop variable.

Example 6.12: Induction Variable Substitution

```
! original code                                    ! transformed code
      DO 10 I=1,10                                       DO 10 I=1,10
         K=1                                                DO 20 J=1,10
         DO 20 J=1,10                                          B(I,J)=B(I,1+3*J)+1
            K=K+2                                  20       CONTINUE
            B(I,J)=B(I,J+K)+1                      10    CONTINUE
20       CONTINUE
10    CONTINUE
```

6.2.2.4 Inline Expansion

Inline expansion textually replaces a procedure call in the program by the associated procedure body, after the actual arguments of the call have been substituted for the dummy arguments, and any necessary syntactic adjustments have been made. Inline expansion can be applied to an acyclic call graph and, in general, leads to more precise results of program analysis. However, it must be applied selectively in order to avoid exponential growth of the program size.

The program information set is **not** automatically updated after inline expansion.

Example 6.13: Inline Expansion

```
! original code                              ! transformed code
      PROGRAM MAIN                                 PROGRAM MAIN

      REAL A(1000),C(1000),E(1000)                 REAL A(1000),C(1000),E(1000)
                                                   INTEGER ARGEXP,ARGE00,ARGE01,OFFDIM
      DO L=1,100
         A(2*L-1)=4*C(L)                           DO L=1,100
         CALL ISDF( A(-1+2*L), E(L), C(L) )          A(-1+2*L)=4*C(L)
      END DO                                         ARGEXP=-1+2*L
      END                                            ARGE00=L
                                                     ARGE01=L
      SUBROUTINE ISDF( X, Y, Z )                     OFFDIM=ARGEXP-1
                                                     E(ARGE00)=A(1+OFFDIM)**3-5*A(2+OFFDIM)
      REAL X(2),Y,Z                                  C(ARGE01)=3*A(1+OFFDIM)**2-5
                                                   END DO
      Y=X(1)**3-5*X(2)                             END
      Z=3*X(1)**2-5
      RETURN
      END
```

6.2.2.5 Dead Code Elimination

Dead code elimination deletes assignment statements to scalar variables which are not live.

Example 6.14: Dead Code Elimination

```
! original code                             ! transformed code
      P=11
      DO 10 I=1,10                                DO 10 I=1,10
         DO 20 J=1,10                                DO 20 J=1,10
            B(I,J)=B(I,11)+1                            B(I,J)=B(I,11)+1
20       CONTINUE                           20       CONTINUE
10    CONTINUE                              10    CONTINUE
```

6.2.2.6 Loop Based Transformations

The *loop based transformations* of VFCS include:

- loop distribution
- loop fusion
- loop interchange
- loop unrolling
- unroll jam
- strip mining
- reduction

They are discussed in the subsections below.

Loop Distribution

Loop distribution is a transformation that distributes the control of a do-loop over groups of statements in its body. It can be applied whenever there exists no backward dependence in the loop.

All `DO`-loops that can be mapped to vector statements are marked as *parallel*. Whenever a loop contains more than one statement, loop distribution must be applied before vectorization can be performed. Loop distribution, when applied to EXSR statements in a loop, can prepare the optimization (vectorization) of communication for that loop.

Example 6.15: Loop Distribution

```
! original code                             ! transformed code
      DO 10 I=1,10                                DO 11 I=1,10                <---  parallel
         C(2*I)=5                                    C(2*I)=5
         DO 20 J=1,10                       11    CONTINUE
            B(I,J)=C(2*I)+A(I,J-1)                DO 12 I=1,10
            A(I,J)=B(I-1,J)+1                        DO 21 J=1,10             <---  parallel
20       CONTINUE                                       A(I,J)=B(I-1,J)+1
10    CONTINUE                              21       CONTINUE
                                                     DO 22 J=1,10             <---  parallel
                                                        B(I,J)=C(2*I)+A(I,J-1)
                                            22       CONTINUE
                                            12    CONTINUE
```

Vector Code Generation

Vector code generation is a transformation that transforms a do-loop into a vector statement. Vector code generation is usually preceded by loop distribution, since it requires that the loop that should be transformed contains exactly one statement.

The following example shows the effect of vector code generation using the code fragment obtained after loop distribution in the previous example.

Example 6.16: Vector Code Generation

```
! original code                            ! transformed code
      DO 11 I=1,10                               C(2:20:2)=5
         C(2*I)=5                                DO 12 I=1,10
11    CONTINUE                                      A(I,1:10)=B(I-1,1:10)+1
      DO 12 I=1,10                                  B(I,1:10)=C(2*I)+A(I,0:9)
         DO 21 J=1,10                      12    CONTINUE
            A(I,J)=B(I-1,J)+1
21       CONTINUE
         DO 22 J=1,10
            B(I,J)=C(2*I)+A(I,J-1)
22       CONTINUE
12    CONTINUE
```

Loop Fusion

Loop fusion is a transformation that combines two adjacent loops with identical ranges into one loop. It can be legally applied if the dependence relation of the program is not modified.

Example 6.17: Loop Fusion

```
! original code                            ! transformed code

      DO 10 I=1,100                              DO 10 I=1,100
         A(I)=A(I)+1                                A(I)=A(I)+1
10    CONTINUE                                      B(I)=B(I)+1
                                           10    CONTINUE
      DO 20 I=1,100
         B(I)=B(I)+1
20    CONTINUE
```

Loop Interchange

Loop interchange is a transformation that exchanges two levels of a nested loop. It can be legally applied iff the dependence relation is kept invariant. Loop interchange is a powerful transformation that can be used to move a parallelizable loop to the right level, to allow vector code generation, to improve the cache behavior of an algorithm and to reduce the number of bank conflicts caused by memory accesses.

Example 6.18: Loop Interchange

```
! original code
      DO 10 I=1,10
         DO 20 J=1,10
            DO 30 K=1,10
               C(I,J)=C(I,J)+A(I,K)*B(K,J)
30          CONTINUE
20       CONTINUE
10    CONTINUE
```

```
! transformed code
      DO 10 I=1,10
         DO 30 K=1,10
            DO 20 J=1,10
               C(I,J)=C(I,J)+A(I,K)*B(K,J)
20          CONTINUE
30       CONTINUE
10    CONTINUE
```

Loop Unrolling

Loop unrolling refers to the process of making one or more copies of the loop body, whereby the loop control must be suitably modified. This can be used to reduce the control overhead for the sequential execution of the loop or to decrease the amount of data movement in a memory hierarchy.

Example 6.19: Loop Unrolling

```
! original code
      DO 10 I=1,1000
               A(I)=B(I+2)*C(I-1)
10    CONTINUE
```

```
! transformed code (unrolling with depth 2)
      DO 10 I=1,999,2
               A(I)=B(I+2)*C(I-1)
               A(I+1)=B(I+3)*C(I)
10    CONTINUE
```

Unroll and Jam

Unroll and jam unrolls an outer loop and then fuses the inner loops back together. This transformation can improve register allocation and cache performance by shortening distances between the sources and sinks of dependences. It will only be performed in the absence of loop carried dependences.

Example 6.20: Unroll and Jam

```
! original code

DO J1D=1,200
  DO I2D=1,1000
    A(I2D,J1D)=A(I2D,1+J1D)+A(I2D,(-1)+J1D)
  END DO
END DO
```

```
! transformed code

   J00 = MOD(200,2)
   DO I00=0,99
     J1D = I00 * 2 + 1 + J00
     DO I2D=1,1000
       A(I2D,J1D)=A(I2D,1+J1D)+A(I2D,(-1)+J1D)
       A(I2D,J1D+1)=A(I2D,2+J1D)+A(I2D,J1D)
     END DO
   END DO
```

Strip Mining

Strip mining (or loop sectioning) refers to a transformation that blocks the iteration range of a loop to exploit locality by introducing an additional loop in the code. Examples include blocking with respect to the boundaries of a local segment, vector register length, or cache size.

Example 6.21: Strip Mining

```
! original code
DO J1D=1,200
  DO I2D=1,1000
    A(I2D,J1D)=A(I2D,1+J1D)+A(I2D,(-1)+J1D)
  END DO
END DO
```

```
! transformed code (with block length 10)
  DO J1D=1,200
    DO I00=0,100
      I01 = I00 * 10 + 1
      DO I2D=I01,MIN(1000,10-1+I01)
        A(I2D,J1D)=A(I2D,1+J1D)+A(I2D,(-1)+J1D)
      END DO
    END DO
  END DO
```

Reduction

Reduction recognition identifies loop nests which represent one of the following reduction functions:

- summing all elements of a vector
- multiplying all elements of a vector
- computation of the dot-product of two vectors
- computation of the maximum value of a vector
- computation of the minimum value of a vector
- determining if all elements of a boolean vector have the value **true**
- determining if there exists an element in a boolean vector that has the value **true**

In case of a successful match, the `DO`-loop is marked with a reference to the recognized reduction function. The actual substitution is performed in the back end based upon that marking.

Example 6.22: Reduction

```
! original code
      S = A(1)
      DO 10 I = 2,100
         IF (A(I) .LT. S) S = A(I)
  10  CONTINUE
```

```
! transformed code
      S = A(1)
      S00 = MINVAL(A(2:100))
      S = MIN(S00,S)
```

Chapter 7

VFCS Performance Tools

In this chapter we describe the performance tools of the Vienna Fortran Compilation System. This includes the *Weight Finder* , a profiler for sequential programs, and P^3T , a performance estimator for data parallel programs.

7.1 Profiling

7.1.1 Introduction

This section describes the *Weight Finder*, a profiler for programs accepted by VFCS (see A). The *Weight Finder* is an integrated tool of the VFCS. The principle aim of this performance analysis tool is to obtain concrete profile data for program unknowns and to find the run-time intensive program portions based on a concrete Fortran 77 program for a characteristic set of program input data. The derived profile data provide helpful information about the underlying program to compilers and performance estimators such as the P^3T.

The source program is loaded. The user chooses out of a set of *sequential program parameters*: frequency information, true-ratios, loop iteration counts and timing information to be derived by the Weight Finder for specific program parts. The Weight Finder then instruments the program according to the chosen sequential program parameters, optimizes, compiles and executes it. As instrumentation code may obviously induce a severe overhead in the profile run, and as a consequence also influences the accuracy of profiled times, emphasis is paid in optimizing the instrumented program. Three optimization techniques are presented which improve the profile run in terms of instrumentation code, profile memory and run-time. The derived sequential program parameters are then visualized on the screen and the internal data structures of the program (syntax tree) are attributed with these values.

The user must provide a characteristic set of input data for the profile run. All derived profile data depend on this input data set.

7.1.2 The Weight Finder and Sequential Program Parameters

In what follows we present a brief description of the *sequential program parameters* which can be optionally determined by a profile run on a von Neuman architecture.

- True ratios (branching probabilities) are determined for conditional statements. True ratios are indicated as values between 0.0 and 1.0, where 0.0 means that the condition is never satisfied during a single profile run of the program, and 1.0 means that the condition is always satisfied.

- Loop iteration counts are determined for DO loops only. This parameter specifies how many times the loop body is executed during a single profile run of the program.
- Frequency information of a statement specifies how many times the statement is executed during a single profile run of the program.
- Timing information can be determined for the overall program, procedures, procedure calls and DO loops. Timing information for a program segment represents the accumulated time this segment was occupying the CPU of the underlying machine.

Note that all but the last parameters are machine independent.

In the following we describe how to use the Weight Finder as an integrated tool of VFCS.

1. **Loading Source Program**
   The program, for which profile information is desired, is loaded under the restrictions of VFCS.
2. **Choosing Sequential Program Parameters**
   The user optionally chooses the *sequential program parameters* of interest:
   - *Restricted Frequency* measures the frequency of all procedure calls and all DO-loops except for perfectly nested DO-loops, for which only the outermost loop is measured.
   - *All Frequencies* measures the frequency of all basic blocks, procedure calls and DO-loops. All loops of a perfectly nested DO-loop are measured.
   - *True Ratios* calculate the overall true-ratio in percentage for all conditional statements.
   - *DO-loop parameters* calculate the average values for DO-loop upper and lower bounds. This is only done for non constant loop bounds.
   - *Time measurement (DO-loops)* measures the accumulated execution time for all DO-loops. For perfectly nested DO-loops only the outermost loop is considered.
   - *Time measurement (procedures)* measures the accumulated execution time for all procedure calls.

   There is an additional option which decreases the amount of instrumentation code and the profile run-time.

   - *Loop parameter check* can be turned on or off. If it is turned on, then instrumentation code is inserted to verify whether or not the loop upper bound is larger than the loop lower bound. In such a case a different instrumentation strategy has to be applied which is more complex and induces higher profile costs in terms of instrumentation code and profile run-time. If the user can guarantee that the loop upper bounds are always larger then the lower bounds then a considerable less complex instrumentation policy is utilized by the Weight Finder. This option is critical as instrumentation code inside of nested loops is expensive.

   The parameter selection is stored every time the user terminates a session under VFCS. After re-starting the same set of parameters and Weight Finder options as chosen at the most recent user session is pre-selected by the system per default.

3. **Start Profiling and Execute**
   The user invokes automatic instrumentation, optimization, profiling and execution under the Weight Finder. The profile data is displayed together with the program in VFCS program window.

7.1.3 How to handle I/O for profile run ?

Profiling a program implies a real execution of the input program. Profiling programs under Weight Finder automatically handles opening files and associated **I/O** operations of the input program. However, if the program requires interactive I/O then the input data has to be written to a file named **WF_INPUT** before starting the profile run. Data which is written by the instrumented program to standard I/O during the profile run is retargeted to **WF_OUPUT**.

7.1.4 Limitations

- For each program change, e.g. problem size or program transformation the profile run has to be redone in order to update the sequential program parameters.
- The *Weight Finder* is able to process both Vienna Fortran as well as High Performance Fortran (HPF) programs. The *Weight Finder* ignores all explicit parallel statements of Vienna Fortran and HPF. Please note that message passing programs (containing send/receive calls) cannot be processed by Weight Finder.

7.2 Snapshot Session for the Weight Finder

In this section we illustrate a snapshot session comprising 2 snapshots, for an HPF JAKOBI version under the *Weight Finder*.

The *VFCS* user interface implementation is based upon X Windows and OSF Motif. When the *VFCS* is started, the **Main Window** (see Figure 7.1) with the title bar **Vienna Fortran Compilation System** appears on the screen. The window is subdivided into two areas. In the **Program Window** the user is able to see the current (possibly transformed) state of the chosen program. Below the program window there is the **Status Window**, which informs the user about what the *VFCS* is currently doing and provides further information on the activities of the system.
In the following we provide a brief description for each of the 2 snapshots in chronological order:

1. The **Performance** menu contains an item for the *Weight Finder*. Selecting the pulldown sub-menu *Performance→Weight Finder→Sequential Program Parameters* pops up the **Sequential Program Parameters** selection box (see Figure 7.1). This box allows to select the sequential program parameters for which the *Weight Finder* obtains characteristic values. We select all but the *restricted frequencies* entry.
2. By using the mouse we single-click on *Performance→Weight Finder→Start Profiling*

   Attribute to initiate the creation of an optimized instrumented program (see the program window), and thereafter to compile and execute it. The *VFCS* returns with an **Information Box** (see Figure 7.2) to report to the user that the profile execution is done. Finally, the program is attributed with the obtained sequential program parameters. In addition the parameter values are visualized on the right hand side of each program statement. FR specifies frequencies, T measured times, $LP = [lb, ub]$ the average lower and upper bounds for the associated loop variable, and TR the true ratio.

7.3 Performance Prediction

This section describes P^3T, a static parameter based performance prediction tool for a subset of Vienna Fortran and HPF programs. P^3T is an integrated tool of VFCS. The main purpose of the P^3T is to provide the programmer with performance feedback on a parallel program version without executing or simulating the code. In order to examine whether a specific program transformation or a data distribution strategy causes a slowdown or improves the performance of a parallel program, the P^3T ([13, 12]) automatically computes

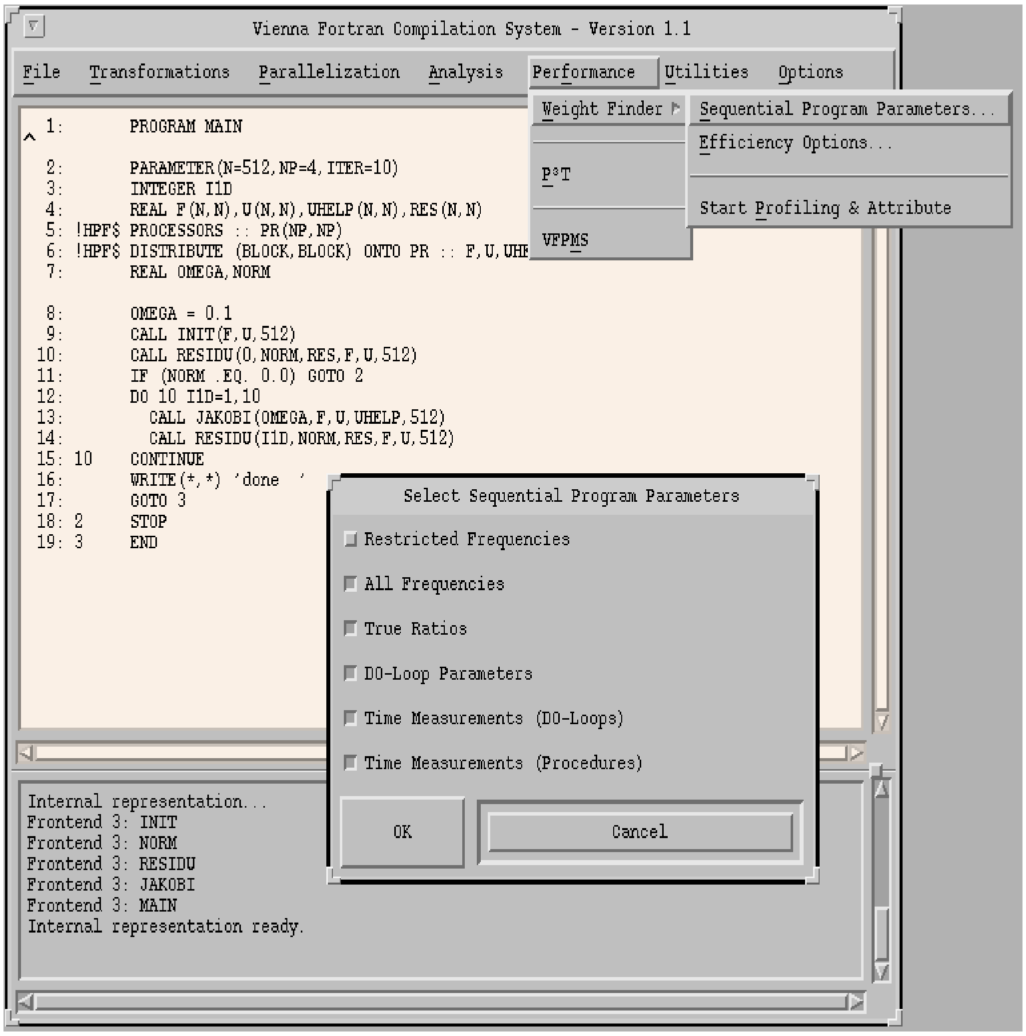


Figure 7.1: HPF Jakobi Version Before Choosing Sequential Program Parameters

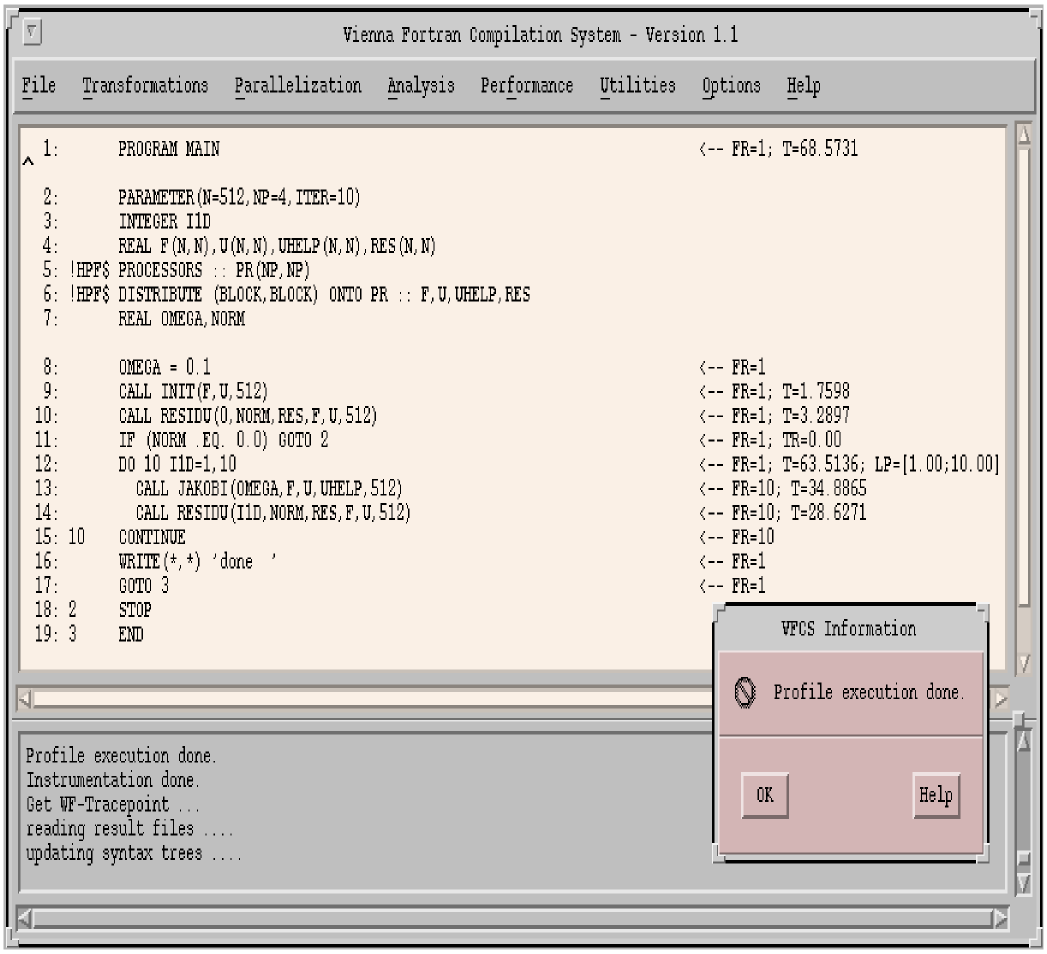


Figure 7.2: After profile is done.

at compile time a set of parallel program parameters. This includes: work distribution, number of transfers, amount of data transferred over the network, transfer times, network contention, number of cache misses, and computation time overhead. In the following the parallel program parameters as computed by the P^3T are briefly described. They are defined and discussed in detail in [13, 12].

- *Work Distribution (WD)*: estimates how even the work contained in a parallel program is distributed across all processors executing the program. The work distribution is inherently defined by the owner computes paradigm. A work distribution value 0 indicates best-case work distribution, whereas increasing values indicate deteriorating work distribution.
- *Number of Transfers (NT)*: is an estimate of the number of send/receive operations induced by a communication statement.
- *Amount of Data Transferred (TD)*: approximates the data volume (in bytes) transferred over the communication network as implied by a communication statement.
- *Network Contention (NC)*: represents an upper bound of the number of channel contentions[1] induced by a communication statement on the underlying physical communication network.
- *Transfer Time (TT)*: predicts the time needed to transfer the required non-local data over the physical communication network.
- *Number of Cache Misses (CM)*: models the data locality of program statements. It is defined by an upper bound of the number of accessed cache lines, which is assumed to correlate with the number of cache misses.
- *Computation Time (CT)*: is an estimate for the sequential computation time of the parallel program. This parameter does not account for communication or blocking (waiting) time. It is very useful to reflect sequential computation overhead implied by the parallelization. E.g. additional IF-THEN-ELSE sequences to implement the owner computes paradigm or more complex loop headers to guarantee that every processor executes only local loop iterations.

The parameters are adapted such that they can be selectively computed for single processors, statements, loops, procedures and the entire program.

For all parameters the following holds: A parameter value 0 implies the best case performance outcome, while increasing values indicate a performance degradation. E.g. if the number of transfers is 0, then there is no communication at all, which is the best case. In most cases the communication overhead deteriorates with increasing number of transfers. The theoretic worst case work distribution outcome is given by $|P| - 1$, where P is the set of processors incorporated to parallelize a given code.

7.3.1 How to use P^3T

In this section we describe the context in which the P^3T is applied in the *VFCS*. Moreover, the graphically oriented user interface of the P^3T is presented.

1. *Load Source Program*
   In the first step, the source program is loaded by *VFCS* .

[1] It is assumed that a channel contention occurs if a physical network channel is traversed by two messages at the same time in the same direction.

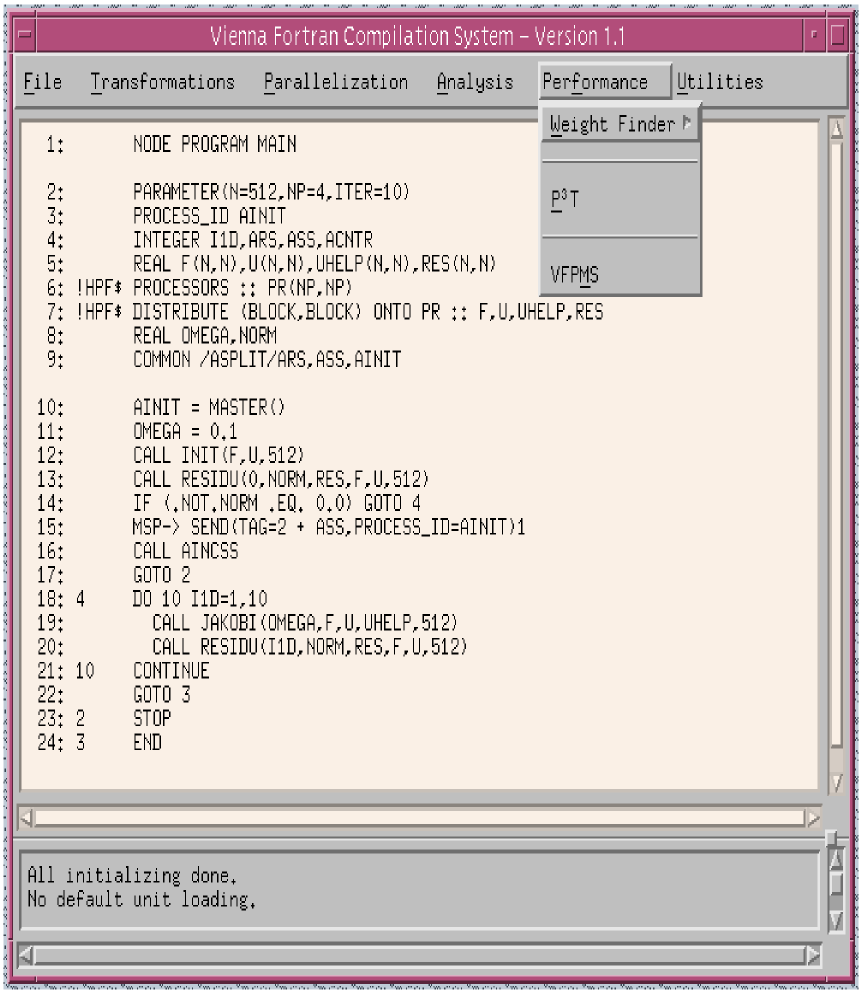


Figure 7.3: VFCS main window after optimizing communication and masking transformations

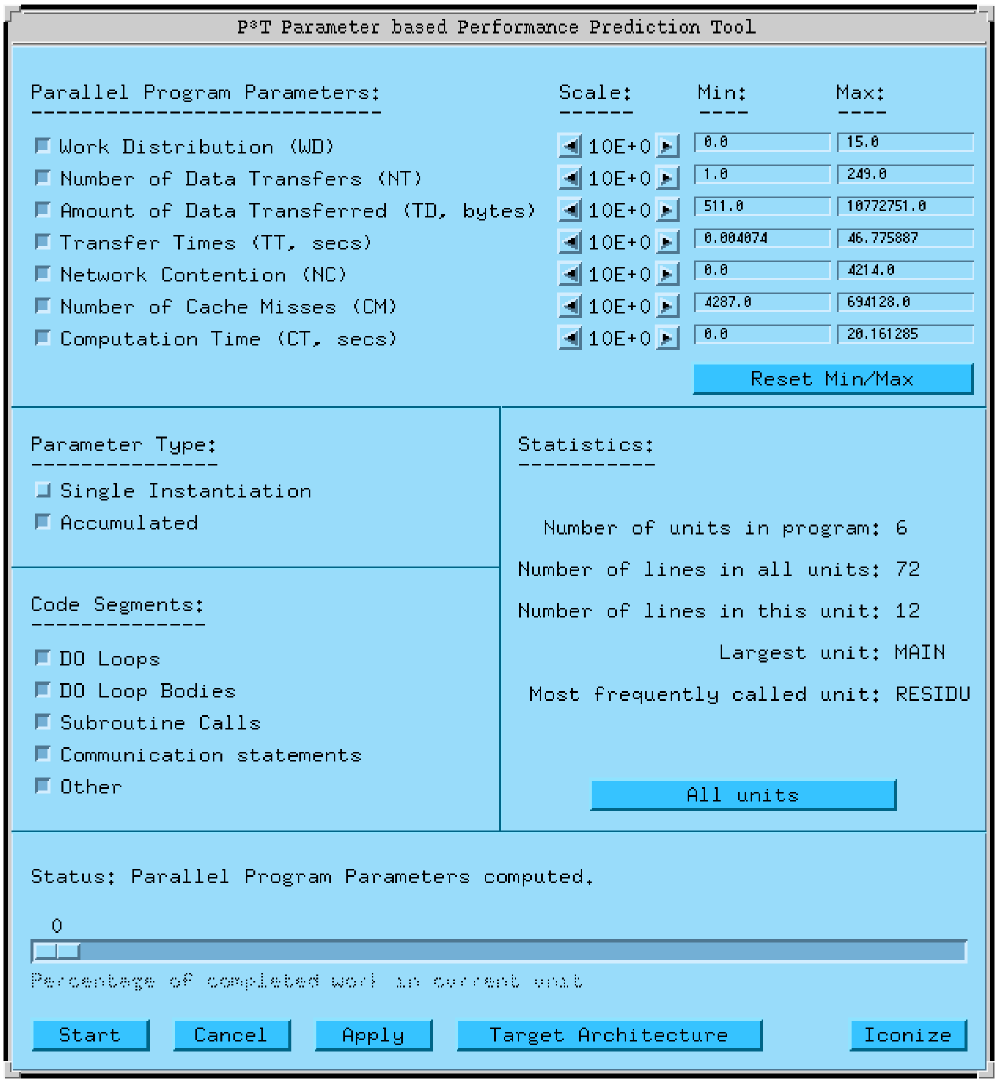


Figure 7.4: This window enables the user to select the parallel program parameters for specific code segments of interest. The resulting parameters can be scaled by a fixed order of magnitude and computed based on accumulated or single execution instance values.

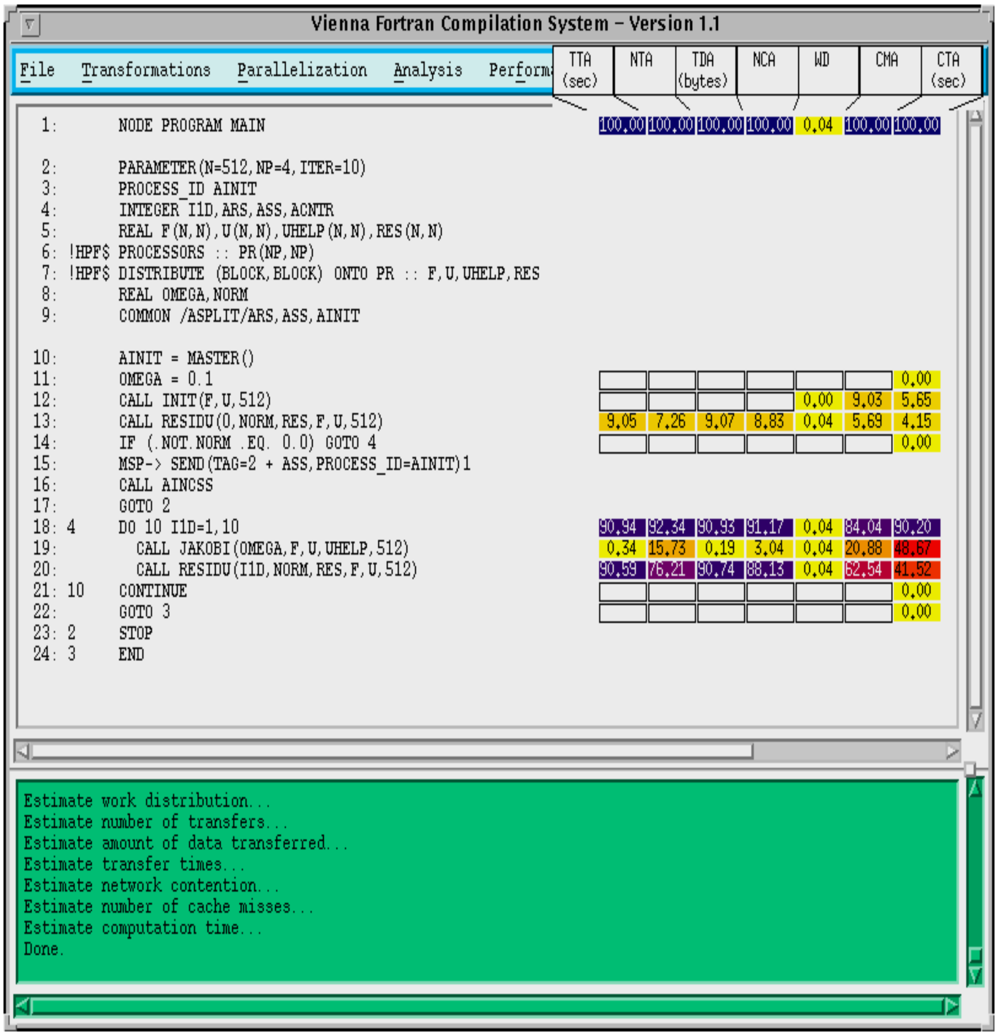


Figure 7.5: HPF JACOBI main program version with P^3T performance data in *VFCS* main window. Performance data is displayed by color-coded performance bars (set of sub-bars) for each program statement of interest. Each sub-bar corresponds to a specific parameter outcome: transfer times (TTA), number of transfer (NTA), amount of data transferred (TDA), network contention (NCA), work distribution (WD), number of cache misses (CMA), and computation time (CTA). The performance goodness is indicated by a range of colors; the brighter the color the better the performance outcome. Dark colors reflect performance losses (hot spots). Each sub-bar contains a percentage value which indicates the performance outcome relative to the worst performance outcome found in the entire program. Blank sub-bars indicate non-applicable performance parameters – for instance, pure computation statements cannot contain any communication.

2. *Weight Finder*
In the next step, the Weight Finder (see Section 7.1 - [14]), which is an advanced and highly optimized profiler for Fortran programs, must be used to determine characteristic values for the following program unknowns: true ratios, loop iteration counts and frequencies. Note that P^3T cannot be applied to any code without previously invoking the Weight Finder on this code for the program unknowns mentioned. Choose the sequential program parameters: "All Frequencies", "True Ratios", and "DO-loop parameters" under the Weight Finder selection box. Note that the profile run is done based on a sequential program. Therefore, for the profile run all explicit parallel constructs are ignored (for instance, PROCESSORS and DIST statements) or replaced (for instance, DOINDEPENDENT by DO loop).

3. *Parallelization and Optimization*
Based on the program attributed by the concrete values for program unknowns, the user may select a data distribution and/or a sequence of program transformations to parallelize and optimize the program. A data distribution imposes the parallelism on a program, while program transformations such as loop interchange and distribution are used to tune a program regarding cache performance, communication overhead and load balance. Figure 7.3 shows the Vienna Fortran main window after the transformations "optimize communication" and "optimize masking" under the *Parallelization menu* has been invoked.

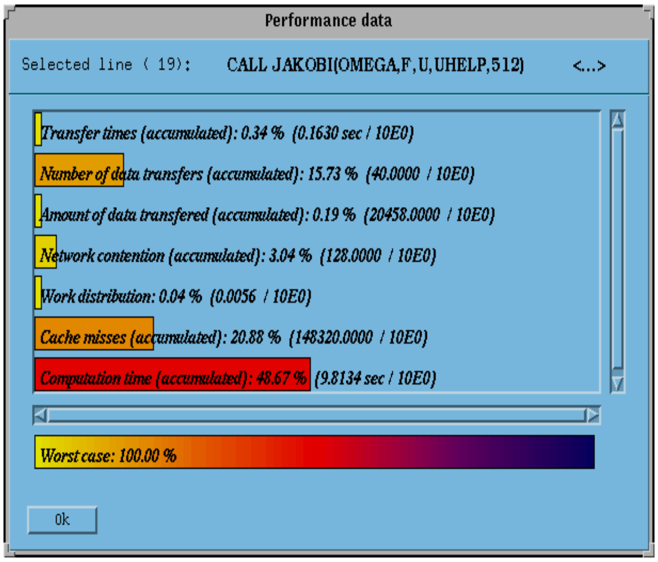


Figure 7.6: Performance visualization for a single statement. For each parameter the precise absolute outcome is visualized. The worst case performance outcome is established by the color-coded bar at the bottom of this window. All parameter bars are compared to this bar. In addition to the color also the length of the bar reflects the performance goodness, which is indicated relative to the worst performance outcome found in the entire program.

4. P^3T
Selecting the P^3T entry in the performance pulldown menu of the *VFCS* main window (Figure 7.3) presents the P^3T *Options* window (Figure 7.4). This window enables the user to select the parallel program parameters for various code segments (for instance, DO loops, communication statements, etc.) of interest. The resulting parameter values can be scaled by multiplying them with a factor $v \in \{10^{-9}, \ldots, 10^0, \ldots, 10^9\}$. This can be useful for large parameter values such as number of cache misses, or if the amount of data transferred should be displayed as Mbytes instead of bytes.

After the parallel program parameters are computed by the P^3T, the minimum and maximum values for each parameter is indicated at the upper right portion of this widget. The minimum and maximum values represent the best and worst found performance outcome, respectively. All performance data are displayed relative to these values. A given minimum – best – performance outcome might be not a good performance figure in reality. In a similar way also the maximum – worst – performance outcome might not be bad performance behavior. The P^3T allows the user to interactively change the minimum and maximum values to reflect a more realistic best and worst case performance figure.

Selecting the single instantiation parameter type results in the display of the parameters based on a single execution of the selected code segments, while the accumulated type yields accumulated figures regarding the entire program run.

Selecting the *Start* button in the P^3T Options window initiates the computation of the parallel program parameters by the P^3T. Based on the parallel program parameters the user is able to identify the location and the nature of performance losses. Figure 7.5 shows the *VFCS* main window with color-coded performance bars for all selected source code lines. A performance bar is defined by a sequence of sub-bars, each of which corresponds to a specific parameter outcome. TTA, NTA, TDA, NCA, WD, CMA, and CTA respectively specify the accumulated parameter outcome for transfer times, number of transfer, amount of data transferred, network contention, work distribution, number of cache misses, and computation time overhead. The performance goodness is indicated by a range of colors; the brighter the color the better the performance outcome. Dark colors reflect performance losses (hot spots). In addition each sub-bar contains a percentage value which is compared to the worst case performance outcome of the entire program. For instance, the *TDA* value (0.19 %) of source code line 19 indicates that this line accounts for 0.19 % of the accumulated amount of data transferred with respect to the entire program. The color-coded bar of the first source code line (*NODE PROGRAM MAIN*) corresponds to the performance of the entire program.

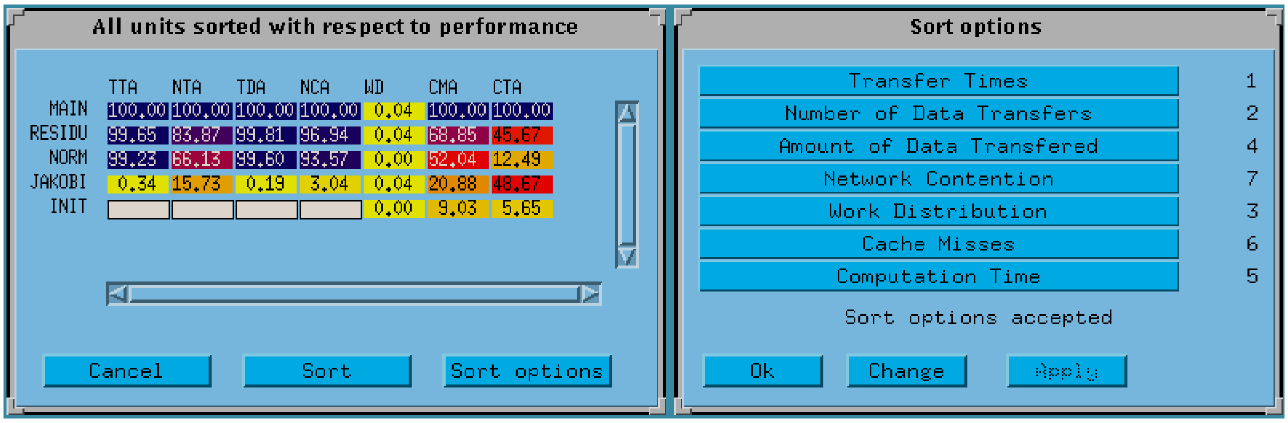


Figure 7.7: Sorted list of program units with respect to performance. The sorting is based on a list of parameter priorities, which can be optionally specified via the sort options window. Each parameter can be given a priority value between 1 and 7, where 1 defines the highest and 7 the lowest priority, respectively. As a consequence, the MAIN program reflects the worst and the INIT subroutine the best performance outcome.

Clicking on an arbitrary source code line – for instance, line 19 in Figure 7.5 – will present a *Performance Data* window (cf. Figure 7.6), which visualizes detailed performance information for that line. For each parameter the precise absolute outcome is visualized. The worst case performance outcome is established by the color-coded bar at the bottom of this window. All parameter bars are compared to this bar. In addition to the color also the length of the bar reflects the performance goodness.

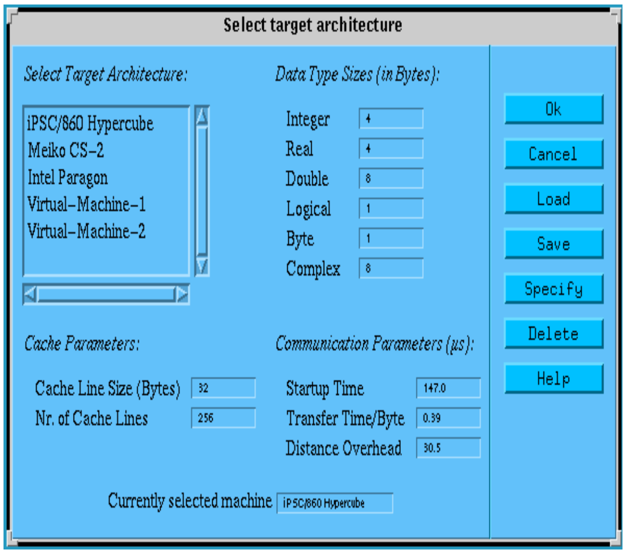


Figure 7.8: This window allows the user to change target architecture specific parameters, on which the computation of the parallel program parameters is based on. The user can either pre-select a target architecture (e.g. iPSC/860) or create a virtual machine by arbitrary defining the machine parameters and storing them under a virtual machine name. Architecture parameters can be changed and virtual machines can be stored, loaded and deleted.

A global performance overview is presented by selecting the *All Units* button in the P^3T *Options* window. A window (Figure 7.7) pops up, which displays a sorted list of all program units[2] with respect to performance. The sorting is based on a list of parameter priorities, which can be specified by the programmer using the “Sort options” widget. After pressing the button “Change” the user can click on the parameter buttons. The first parameter button clicked on indicates the highest priority for the sake of sorting. The second parameter clicked on means the second highest parameter priority, and so on. By clicking on the “Apply” button, the user verifies the sort options. Subsequent clicking on the “Sort” button of the “All units sorted ...” widget sorts the units according to the chosen parameter priorities under the “Sort options” widget. For this example, the transfer time parameter has highest priority. As a consequence, the MAIN program reflects the worst and the INIT subroutine the best performance outcome.

Some of the parallel program parameters depend on architecture specific factors, such as data type sizes, cache line size, startup time, etc. By choosing “Target Architecture” under the P^3T Options widget a window (cf. Figure 7.8) pops up which allows to choose out of a set of pre-defined architectures such as the iPSC/860 hypercube. The user can also specify its own machine by clicking on “Specify” and consecutively changing the machine parameters. Thereafter, the so chosen private machine can be stored using the “Save” button. “Delete” allows to delete a machine other than those pre-defined by the system. “Load” enables the programmer to load a private machine specification.

5. *Backend*

Once the P^3T reports reasonable performance gains or the code development time limit is reached, the

[2] The main program, all functions and subroutines define a unique program unit.

user selects a specific backend for the desired target architecture, based on which the *VFCS* generates an explicitly parallel and optimized message passing program.

7.3.2 Language Constructs Accepted by P^3T

P^3T processes a subset of Vienna Fortran and HPF. In terms of Vienna Fortran in is able to accept the following constructs:

- PROCESSORS
- RESHAPE
- DIST
- ALIGN

The following HPF directives can be processed by P^3T:

- PROCESSORS
- DISTRIBUTE
- ALIGN

7.3.3 Restrictions of P^3T

The programs which are analyzed by P^3T must satisfy the following conditions:

- only static and regular problems can be handled.
- vector statements and any kind of Fortran90 specific statements are excluded. Fortran77 programs are accepted and processed by P^3T.
- all array index expressions and loop bound expressions must be linear functions of loop variables. Please be sure that array index expressions do not contain other variables than loop variables.
- indirect array accesses (e.g. A(B(I))) and therefore irregular problems cannot be handled
- characteristic values for true ratios (branching probabilities), loop iteration and statement execution counts must be obtained by the Weight Finder before applying the P^3T (see Section 7.1)
- message passing programs cannot be processed by P^3T.
- Currently all program transformations must be applied to the program before the profile run of the Weight Finder is invoked. P^3T cannot yet automatically adapt profile data after applying program transformations. The only code changes that can be legally applied before and after using P^3T is changing the number of processors or data distribution. For these code changes the profile run does not have to be repeated. For changes in problem size and code transformations the profile run must be repeated.
- Enclosing loops whose loop variables do not appear in array index expressions of the associated loop body may occasionally imply error messages. Manual replacing such loops by GOTO loops should prevent this problem.

In order to take procedure calls[3] into account, the parameter outcome for a single procedure call instantiation is supposed to be independent of the call site. This means that the parameter outcome at a particular call site is the same as the parameter outcome of the procedure over all call sites, which is a common assumption made for performance estimators ([47]).

[3] This refers to Fortran subroutine and function calls.

7.4 Vienna Fortran Performance Measurement System

7.4.1 Introduction

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 the Vienna Fortran Compilation System (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 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.

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*.

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.

VFPMS includes a context sensitive help system providing information in every situation within an user session. The history of instrumentation can be easily followed using the *component browser*.

7.4.2 Using VFPMS

Clicking the menu item *VFPMS* in the *Performance menu* pops up the VFPMS main menu as shown in Figure 7.9.

The menu consist of the buttons

- *Time Unit*
- *Time Loop*
- *Time Range*
- *Time Statement*
- *Component Browser*
- *Help*
- *Exit*

for selecting one of the component based instrumentation modes, opening the browser tool, getting help and closing the menu.

7.4.2.1 Using the Component Browser

Clicking the button *Component Browser* starts the browser tool which is shown in Figure 7.9.
The component browser provides information on the instrumentation progress. and is actualized each time a *measuring component* is set, by adding an appropriate *description item* to its window. Clicking at a description item in the browser's window highlights the corresponding measuring component in the VFCS text window. The component browser does not provide information on other units than the actual one. When switching to another unit, the component browser (if opened) is actualized automatically. A *measuring component* can be deleted by pressing the *DELETE* button.

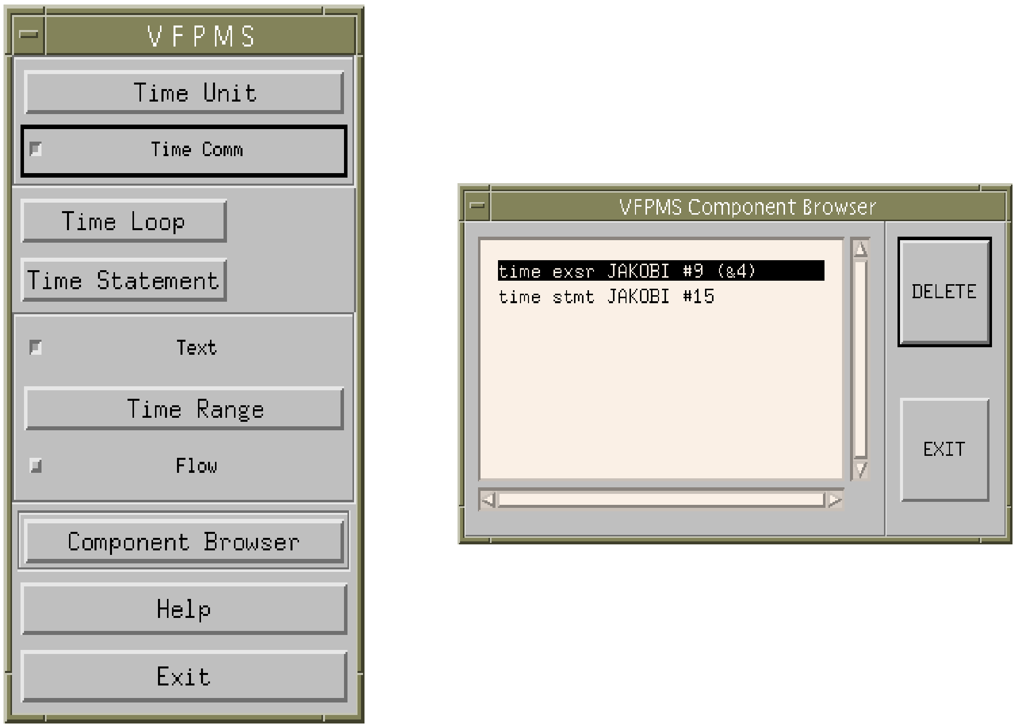


Figure 7.9: The VFPMS Main Menu and the Component Browser

7.4.2.2 Time Measurement of a Program Unit

Clicking at the *Time Unit* button in the component browser causes a measuring point to be set at the beginning and at each exit point of the actual unit. As a result, the total execution time of the corresponding unit will be measured.

In addition to the total execution time of a unit, the communication time of a unit may be measured separately, by switching the *time comm* (toggle) button (located beneath the time unit button) on. Using this feature for the unit `MAIN` causes the measurement of the communication time in every subroutine of the program, too.

7.4.2.3 Time Measurement of a Loop

Clicking the *Time Loop* button opens a message box which requests the user to select a `DO statement`. This can be done using the mouse or the arrow keys. After clicking on the *OK* button of the message box the loop component will be set. Alternatively, the user can cancel the action by clicking on the *CANCEL* button or get help information by clicking on the *HELP* button of the message box.

7.4.2.4 Time Measurement of a Code Range

Setting a code range for performance measurement can be achieved, in two steps as follows. Clicking at the *Time Range* button in the VFPMS main window, a message box is displayed, asking the user for selecting the first statement of the code range to be measured. Now, the user clicks at the first statement *r0* of the code range and then clicks the *OK* button of the message box. After that, a second message box request the user to select the last statement *r1* of the code range to be measured, which is done in the same way as outlined above.

After successful specification of the code range, the system automatically determines a set of measuring points set on the entries and exits of the specified range, based on either the textual representation or the flow graph of the program.

Text Based Range Measurement

The text based range measurement can be selected by toggling the *Text* button which is located immediately above the *Time Range* button. As a consequence, all the statements which are textually between *r0* and *r1* including *r0* and *r1* belong to the code range.

Flow Graph Based Measurement

The flow graph based range measurement can be selected by toggling the *Flow* button which is located immediately below the *Time Range* button. With this method a range is regarded as the set of paths between *r0* and *r1* based on the flowgraph.

7.4.2.5 Time Measurement of Statement

Setting a single statement component consists in specifying the desired statement and clicking on the OK button of the message box.

Bibliography

[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 (to appear)*, 1992.

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

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

[5] S. Benkner. Vienna Fortran 90 and its Compilation. Ph.D.Thesis. Technical Report, University of Vienna, Institute for Software Technology and Parallel Systems, September 1994.

[6] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, 1988.

[7] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran (To appear: Scientific Programming 1992). ICASE Report 92-9, ICASE, Hampton, VA, 1992.

[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] M. Chen and J. Li. Optimizing Fortran 90 programs for data motion on massively parallel systems. Technical Report YALE/DCS/TR-882, Yale University, January 1992.

[10] M. Chen, Y.Choo, and J. Li. Theory and pragmatics of generating efficient parallel code. In *Parallel Functional Languages and Compilers*. ACM Press and Addison-Wesley, 1991.

[11] R.J. Duffin. On Fourier's analysis of linear inequality systems. *Mathematical Programming Study* 1, 71-95, 1974.

[12] 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

[13] Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers. University of Vienna, Institute of Software Technology and Parallel Systems, October, 1993

[14] 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.

[15] G. Fox et.al. Solving Problems on Concurrent Processors. Prentice Hall, Englewood Cliffs, 1988.

[16] G.Fox, S.Hiranadani, K.Kennedy, C.Koelbel, U.Kremer, C.Tseng, M.Wu. Fortran D Language Specification. Rice University, Technical Report COMP TR90-141, December 1990.

[17] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Department of Computer Science Rice COMP TR90079, Rice University, March 1991.

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

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

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

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

[22] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Proceedings of the The Fifth Distributed Memory Computing Conference*, pages 1105–1114, Charleston, SC, April 1990.

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

[24] K. Kennedy and H. Zima. Virtual shared memory for distributed-memory machines. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, March 1989.

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

[26] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[27] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Semi-automatic process partitioning for parallel computation. *International Journal of Parallel Programming*, 16(5):365–382, 1987.

[28] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory architectures. In *2nd ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, pages 177–186, March 1990.

[29] H.W.Kuhn. Solvability and Consistency for Linear Equations and Inequalities *American Mathematical Monthly* 63, 217-232, 1956

[30] J. Li and M. Chen. Generating explicit communication from shared-memory program references. In *Proceedings of Supercomputing '90*, pages 865–876, New York, NY, November 1990.

[31] J. Li and M. Chen. Compiling Communication-Efficient Programs for Massively Parallel Machines. *IEEE Transactions on Parallel and Distributed Systems* Vol.2(3), 361-376, July 1991.

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

[33] D. Loveman. High Performance Fortran: Proposal. In *High Performance Fortran Forum,* Houston, TX, January 1992.

[34] G. I. Marchuk. *Methods of Numerical Mathematics.* Springer-Verlag, 1975.

[35] P. Mehrotra, J. Van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, Vol. 5, 339-361, 1987.

[36] P. Mehrotra. Programming parallel architectures: The BLAZE family of languages. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 289–299, December 1988.

[37] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using Kali. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 364–384. Pitman/MIT-Press, 1991.

[38] MIMDizer User's Guide, Version 8.0. Applied Parallel Research Inc., Placerville, CA., 1992

[39] E. Paalvast and H. Sips. A high-level language for the description of parallel algorithms. In *Proceedings of Parallel Computing 89*, Leyden, Netherlands, August 1989.

[40] E. Paalvast, A. van Gemund, and H. Sips. A method of parallel program generation with an application to Booster language. In *Proceedings of the 4th International Conference on Supercomputing*, Amsterdam, June 1990.

[41] D. Pase. MPP Fortran programming model. In *High Performance Fortran Forum,* Houston, TX, January 1992.

[42] D.M. Pase, T. MacDonald, A. Meltzer. MPP Fortran Programming Model. Technical Report, Cray Research, March 1994.

[43] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Conference on Programming Language Design and Implementation*, pages 69–80. ACM SIGPLAN, June 1989.

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

[45] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(2):303–312, 1990.

[46] G. Steele. Proposals for amending high performance Fortran. *Communicated at the High Performance Forum*, Houston, January 1992.

[47] V. Sarkar, Partitioning and Scheduling Parallel Programs for Multiprocessor, The MIT Press, Cambridge, Massachusetts,1989

[48] P. S. Tseng. A systolic array programming language. In *Proceedings of the Fifth Distributed Memory Computing Conference*, pages 1125–1130, April 1990.

[49] E. Van de Velde. Experiments with multicomputer LU-decomposition. Technical Report Series CRPC-89-1, California Institute of Technology, April 1989.

[50] J. Wu, J. Saltz, H. Berryman and S. Hiranandani. Distributed memory compiler design for sparse problems. ICASE Report 91-13, January 1991.

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

[52] H. Zima, H. Bast, M. Gerndt, and P. Hoppen. Semi-automatic parallelization of Fortran programs. In *CONPAR 86, Conference on Algorithms and Hardware for Parallel Processing*, volume LNCS 237, pages 287–94. Springer, 1986.

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

[54] 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

[55] P. Brezany, B. Chapman, H.P. Zima. Automatic Parallelization for Genesis (Esprit Project 2702 – Genesis) Technical Report ACPC/TR 92-16, Austrian Center for Parallel Computation (November 1992)

Appendix A

Input Languages

VFCS accepts as input languages Fortran 77 with Fortran 90 extensions and a subset of Vienna Fortran and HPF data distribution features. The following sections describe the syntax of the Vienna Fortran and HPF features that are currently implemented, and describes the available Fortran 90 features. For a detailed semantic description refer to the language specifications.

Notation

The notation used in the syntax rules is a variation of Backus-Naur form and in accordance with the Fortran 90 standard document [23].

The syntactic metasymbols used are:

is	introduces a syntactic class definition
or	introduces a syntactic class alternative
[ ]	encloses an optional item
[ ] ...	encloses an optionally repeated item which may occur zero or more times

Assumed syntax rules are:

xyz-list	**is**	*xyz* [, *xyz*] ...
xyz-name	**is**	*name*
integer-xyz	**is**	*xyz*

Fortran 90 Features

VFCS currently supports the following Fortran 90 features:

- Fortran 90 style of type declaration statements.
- *Array Assignment Statements.*
- *Array Sections* may be used in the context of array assignment statements and for referencing Vienna Fortran processor array sections. The use of vector subscripts is not supported. Array sections must not be actual arguments of a procedure call.
- *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* but without *named DO construct.*

A.1 Vienna Fortran

A.1.1 Intrinsic Distribution Functions

A.1.1.1 BLOCK(M)

Optional Argument. `M`

Description. Determines a regular block distribution.

Class. 1-1 distribution function.

Argument. `M` must be scalar of type integer.

Result Type and Value.

Creates a block distribution. Let P denote the number of processors to which the distribution refers and let N denote the size of the array (dimension) to be distributed.

Case (i): The optional argument M is not specified. The array (dimension) is divided into contiguous blocks, whose sizes differ by at most 1. Let $r := (N \ mod \ P)$, and $q := \lfloor \frac{N}{P} \rfloor$. The array is divided into r blocks of size $q+1$ — which are mapped to the first r processors — and $P-r$ blocks of size q, mapped to the remaining processors.

Case (ii): If the optional argument M is specified explicitly, the array (dimension) is divided into $n = \lfloor \frac{N}{M} \rfloor$ contiguous blocks of size M which are mapped to the first n processors, with $0 \leq n \leq P$. If N is not exactly divisible by M, then the remaining $(N \ mod \ M)$ elements are mapped to processor $n+1$. All other processors (if any) do not receive any elements.

A.1.1.2 B_BLOCK(ARRAY)

Description. Determines a general block distribution.

Class. 1-1 distribution function.

Argument. `ARRAY` must be a rank one array of type integer.

Result Type and Value. Let

- $[1:N]$ denote the index domain of `ARRAY`,
- $[L:U]$ denote the index domain associated with the array (dimension) to be distributed, and
- P denote the number of processors in the processor array (dimension), which is the target of the distribution.

Let P the number of processors in the processor array (dimension), which is the target of the distribution. Then $N \geq P-1$. The array (dimension) is partitioned into P contiguous blocks. For all $i, 1 \leq i < P$, `ARRAY`(i) specifies the upper bound of Block i. The index ranges associated with the blocks are given as follows:

- Block 1: $[L:$`ARRAY`$(1)]$
- Block i, $1 < i < M$: $[$`ARRAY`$(i-1)+1:$`ARRAY`$(i)]$
- Block P: $[$`ARRAY`$(P-1)+1:U]$

The values of *int-array* must be defined in such a way that each block is associated with a nonempty index range.

A.1.1.3 S_BLOCK(ARRAY)

Description. Determines a general block distribution.

Class. 1-1 distribution function.

Argument. `ARRAY` must be a rank one array of type integer.

Result Type and Value. Let

- $[1 : N]$ denote the index domain of `ARRAY`,
- $[L : U]$ denote the index domain associated with the array (dimension) to be distributed, and
- P denote the number of processors in the processor array (dimension), which is the target of the distribution.

Then $N \geq P - 1$. The array (dimension) is partitioned into P contiguous blocks. For all $i, 1 \leq i < P$, `ARRAY`(i) specifies the size of Block i. The index ranges associated with the blocks are given as follows:

- Block 1: $[L : L+\texttt{ARRAY}(1) - 1]$
- Block 2, $[L+\texttt{ARRAY}(1) : L+\texttt{ARRAY}(1)+\texttt{ARRAY}(2) - 1]$
- ...
- Block P: $[\ldots : U]$

The values of `ARRAY` must be defined in such a way that each block is associated with a nonempty index range.

A.1.2 INDIRECT(MAP)

Description. Determines an indirect distribution.

Class. 1-1 distribution function.

Argument. `MAP` must be a rank one array of type integer, and must have the same size as the array(dimension) to be distributed. Each element of `MAP` must designate a valid processor number.

Result Type and Value. Let

- $[1 : N]$ denote the index domain of `MAP`,
- $[1 : N]$ denote the index domain associated with the array (dimension) to be distributed, and let
- P denote the number of available processors.

Each index i of the array (dimension) to be distributed is mapped to processor i, $1 \leq i \leq N$.

A.1.3 Vienna Fortran 77 Syntax

Using the Vienna Fortran 77 syntax [54] distributions and alignments are specified by means of annotations which are appended to array declarations. Furthermore A new executable statement, the **INDEPENDENT** loop[1] is provided.

A.1.3.1 Processor Declarations

vf77-proc-decl-stmt	**is**	*primary-processor-structure* [*secondary-processor-structures*]
primary-processor-structure	**is**	`PROCESSORS` *array-declarator*
secondary-processor-structures	**is**	`RESHAPE` *array-declarator-list*

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

Constraint: A *vf77-proc-decl-stmt* may be specified only once and must occur in the *specification-part* of the main program. (The processor arrays of the main program are visible in all subroutines.)

A.1.3.2 Processor References

processor-reference	**is**	*processor-element-name*
	or	*processor-section* [(/ *dimension-permutation* /)]
processor-element-name	**is**	*array-element-name*
processor-section	**is**	*array-section*
dimension-permutation	**is**	*int-constant-expr-list*
array-section	**is**	*array-name* [(*section-subscript-list*)]
section-subscript	**is**	*subscript*
	or	*subscript-triplet*
subscript	**is**	*scalar-int-expr*
subscript-triplet	**is**	[*subscript*] : [*subscript*] [: *stride*]
stride	**is**	*scalar-int-expr*

A.1.3.3 Declaration Annotations

vf77-type-decl-stmt	**is**	*type-spec entity-decl-list declaration-annotation*
declaration-annotation	**is**	*actual-array-annotation*
	or	*dummy-array-annotation*
actual-array-annotation	**is**	[*distribution-specification*]
	or	[*alignment-specification*]
dummy-array-annotation	**is**	*inherit-annotation*
inherit-annotation	**is**	`DIST ( * )`

[1] **INDEPENDENT** loops correspond to **FORALL** loops as included in the original Vienna Fortran language specification [54].

Constraint: If a *declaration-annotation* is specified all entities of the *entity-decl-list* must be array declarations.

Constraint: Actual arguments that are arrays must not be array sections.

A.1.3.4 Distribution Specifications

distribution-specification	**is**	`DIST` *dspec*
dspec	**is**	`(` *distribution-expr* `)` [`TO` *processor-reference*]
	or	`TO` *processor-reference*
distribution-expr	**is**	*simple-distribution-expr*
	or	*composite-distribution-expr*
	or	*workspace-distribution*
simple-distribution-expr	**is**	*distribution-function-reference*
	or	*distribution-extraction*
distribution-function-reference	**is**	`BLOCK`[`(`*int-expr*`)`]
	or	`CYCLIC`[`(`*int-expr*`)`]
	or	`S_BLOCK(`*expr*`)`
	or	`B_BLOCK(`*expr*`)`
	or	`INDIRECT(`*expr*`)`
distribution-extraction	**is**	`=` *array-or-dimension*
array-or-dimension	**is**	*array-name*
	or	*array-dimension*
array-dimension	**is**	*array-name dimension-qualifier*
dimension-qualifier	**is**	`.` *int-constant-expr*
composite-distribution-expr	**is**	*dimensional-expr-list*
dimensional-expr	**is**	*simple-distribution-expr*
	or	`:`
workspace-distribution	**is**	`WORKSPACE =` *ws-name*

Constraint: The number of *dimensional-expr*s must match the rank of the arrays to be distributed.

Constraint: If a *processor-reference* is not specified, the primary processor array is substituted by default.

Constraint: Any expressions occurring in a *dspec* must be specification expressions.

Constraint: If *workspace-distribution* is used, the corresponding data array(s) to which the distribution refers must have rank 1.

A.1.3.5 Alignment Specifications

alignment-specification	**is**	`ALIGN` *aspec*
aspec	**is**	*alignment-expr*
alignment-expr	**is**	*target-array-identification* (*bound-variable-list*) `WITH` *source-array-reference*
target-array-identification	**is**	*array-name*
	or	`$`
bound-variable	**is**	*variable-name*
	or	:
source-array-reference	**is**	*source-array-name* [(*source-subscript-list*)]
source-subscript	**is**	*subscript*
	or	*subscript-triplet*
	or	*align-subscript*

Constraint: An *align-subscript* must be an *scalar-int-expr* that contains exactly one occurrence of a *bound-variable*.

Constraint: An *align-subscript* must be a specification expression that is linear in the *bound-variable* and contains operators no other than "+","-", and "*".

Note: Vienna Fortran 77 alignments specifications are not yet implemented in VFCS. If you want to use alignment in a Vienna Fortran program, please use the Vienna Fortran 90 syntax as specified in A.1.7.

A.1.4 Workspace Declaration Statement

workspace-decl-statement	**is**	`WORKSPACE` *ws-name* :: *workspace-list*
ws-name	**is**	*name*
workspace	**is**	*array-declarator* `DIST` *dspec*

Constraint: A *dspec* occurring in a *workspace* must not be a *workspace-distribution*.

Constraint: A *dspec* occurring in a *workspace* must not be a an `INDIRECT` distribution.

A.1.4.1 INDEPENDENT Loops

independent-loop	**is**	*independent-statement independent-block end-independent*
independent-statement	**is**	*label-independent-statement*
label-independent-statement	**is**	`DO INDEPENDENT` *label independent-control*
independent-control	**is**	*control-variable* [*on-clause*]
control-variable	**is**	*variable-name* `=` *int-expr* , *int-expr* [, *int-expr*]
on-clause	**is**	`ON` *processor-element-name*
	or	`ON OWNER (` *array-element-name* `)`
independent-block	**is**	*reduction-statement*
	or	*executable-statement*
reduction-statement	**is**	`REDUCE (` *reduction-op* , *variable* , *expression* `)`
reduction-op	**is**	`SUM`
	or	`MULT`
	or	`MAX`
	or	`MIN`

Constraint: No loop-carried dependences are permitted (except of accumulation type dependences)

Constraint: Nested `DO INDEPENDENT` loops are not allowed.

Constraint: There must be at least one access to a distributed array.

Constraint: Distributed arrays in the context of `DO INDEPENDENT` may have only the first dimension distributed.

Constraint: No distributed variable is allowed in subscript functions of an array element used in the ON OWNER *clause*.

Constraint: No input/output, `STOP`, or `RETURN` statements are allowed.

Constraint: No function and subroutine calls are allowed.

Constraint: Scalar variables have the semantics of *private variables* in the sense that they are not live at the end of the `DO INDEPENDENT` loop (except those in a `REDUCE` statement).

A.1.4.2 `DISTRIBUTE` Statement

distribute-statement	**is**	`DISTRIBUTE` *distribution-group*
distribution-group	**is**	*array-list* :: *dspec*

Constraint: In the current implementation a *dspec* appearing in a *distribute statement* may be only used for specifying an indirect distribution.

A.1.5 Vienna Fortran 90 Syntax

Using the Vienna Fortran 90 syntax, distributions and alignments are specified by means of additional attributes within the type declaration.

A.1.5.1 Processor Array Declarations

vf90-proc-decl-stmt	**is**	`PROCESSORS` [[, *proc-attr-spec-list*] ::] *proc-decl-list*
proc-decl	**is**	*proc-array-name* (*proc-shape-spec-list*)
proc-attr-spec	**is**	`DIMENSION` (*proc-shape-spec-list*)
proc-array-name	**is**	*object-name*
proc-shape-spec-list	**is**	*explicit-shape-spec-list*

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

Constraint: *processor-declarations* may occur only in the *specification-part* of the main program. (The processor arrays of the main program are visible in all subroutines.)

A.1.5.2 Processor Array References

proc-ref	**is**	*proc-array-name* [*dimension-permutation*]
	or	*proc-array-section* [*dimension-permutation*]
	or	*proc-array-element*
proc-array-section	**is**	*proc-array-name* (*section-subscript-list*)
proc-array-element	**is**	*array-element*
dimension-permutation	**is**	(/ *int-const-expr-list* /)
section-subscript	**is**	*subscript*
	or	*subscript-triplet*

A.1.5.3 Type Declaration Statement

vf90-type-decl-stmt	**is**	*type-spec* [[, *vf-attr-spec*] ... ::] *entity-decl-list*
vf-attr-spec	**is**	`DIMENSION` (*array-spec*)
	or	*dist-attr-spec*
	or	*align-attr-spec*
array-spec	**is**	*explicit-shape-spec-list*

Constraint: If *vf-attr-spec* is present all entities of the *entity-decl-list* must designate arrays.

Constraint: If *dist-attr-spec, align-attr-spec*, attribute is present, *type-spec* must not be `CHARACTER`.

A.1.6 The DISTRIBUTED Attribute

dist-attr-spec	**is**	`DISTRIBUTED` *dist-spec*
	or	`DISTRIBUTED TO` *proc-ref*
dist-spec	**is**	`(` *dist-type-spec* `)` [`TO` *proc-ref*]
	or	*assumed-dist-spec*
dist-type-spec	**is**	*dim-dist-type-spec-list*
	or	*general-dist-type-spec*
dim-dist-type-spec	**is**	*dim-dist-func-ref*
	or	*dist-extraction*
	or	:
general-dist-type-spec	**is**	*gen-dist-func-ref*
	or	*dist-extraction*
dim-dist-func-ref	**is**	`BLOCK`[(*int-expr*)]
	or	`CYCLIC`[(*int-expr*)]
	or	`S_BLOCK`(*expr*)
	or	`B_BLOCK`(*expr*)
	or	`INDIRECT`(*expr*)
gen-dist-func-ref	**is**	`INDIRECT`(*expr*)
dist-extraction	**is**	=*variable-name*[.*dimension-qualifier*]
dimension-qualifier	**is**	*int-expr*
assumed-dist-spec	**is**	`( * )`

Constraint: The number of *dim-dist-type-spec*s must match the rank of the array.

Constraint: An *assumed-dist-spec* may be specified for dummy arrays only.

Constraint: If a *proc-ref* is not specified, the primary processor array is substituted by default.

Constraint: Any expressions occurring in a *dist-attr-spec* must be specification expressions.

Constraint: The argument of *S_BLOCK* and *B_BLOCK* must be a one-dimensional integer array expression.

Constraint: A *dimension-qualifier* must be an integer constant expression.

A.1.7 The ALIGNED Attribute

align-attr-spec	**is**	`ALIGNED` *align-spec*
align-spec	**is**	*element-align-spec*
	or	*section-align-spec*
element-align-spec	**is**	(*alignee-spec* `WITH` *align-source-spec*)
alignee-spec	**is**	[*alignee-name*] (*dim-alignee-spec-list*)
dim-alignee-spec	**is**	*bound-variable-name*
	or	:
align-source-spec	**is**	*align-source-object* [(*source-subscript-list*)]
source-subscript	**is**	*subscript*
	or	*subscript-triplet*
	or	*align-subscript*
align-subscript	**is**	*scalar-int-expr*
section-align-spec	**is**	([[*alignee-name*] `WITH`] *align-source-section*)
align-source-section	**is**	*align-source-object* [(*section-subscript-list*)]
alignee-name	**is**	*variable-name*
align-source-object	**is**	*variable-name*

Constraint: The number of *dim-alignee-specs* must match the rank of the alignee.

Constraint: A *bound-variable* occurring in an *align-spec* is implicitly declared of type integer. Any variable in the corresponding scoping unit with the same name is unaffect by it. The scope of the bound variable is restricted to the *align-spec*.

Constraint: *subscript* and *subscript-triplet* must not contain a bound variable.

Constraint: An *align-subscript* must be an scalar-int-expr that contains at exactly one occurrence of a *bound-variable*.

Constraint: An *align-subscript* must be a specification expression that is linear in the *bound-variable* and contains operators no other than "+","-", and "*".

Constraint: If *alignee-name* is not present, the *ALIGNED* attribute is applied to all arrays in the *entity-decl-list*.

Constraint: The *align-source-section* must have the same rank as the *alignee*.

Constraint: All expressions occurring in an *align-spec* must be specification expressions.

A.2 HPF

In HPF distributions and alignments are specified by means of directives.

A.2.1 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.2.2 PROCESSORS Directive

processors-directive	**is**	`PROCESSORS` *processors-decl-list*
processors-decl	**is**	*processors-name* [(*explicit-shape-spec-list*)]
processors-name	**is**	*object-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.2.3 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	`*`
dist-onto-clause	**is**	`ONTO` *dist-target*
dist-target	**is**	*processors-name*

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.

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.2.4 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	*subscript-triplet*
	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.2.5 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.

Appendix B

Intermediate Language

The effects of the parallelizing transformations and user-selectable transformations on the Fortran source code can be observed in the VFCS program window. Some effects result in modification to the source code itself; other information is provided by a number of internal notations which are not processed into Fortran code until the parallelization session is completed. We describe this notation here.

B.1 Masks

VFCS uses two classed of masks, *statement masks* and *communication masks*. Statement masks describe the ownership of distributed data in order to enforce the owner computes rule. Communication mask indicate which processor is responsible for communicating data.

B.1.1 Statement Masks

Masked Assignment Statements

Assignment statements where the left-hand-side variable is distributed are prefixed by the following mask:

OWNED (*array element***) ->**

As a consequence, the statement is only executed by those processors which are owners of *array element*. This usually restricts the number of processors which execute the statement. (See also Section 4.4.3).

Masked DO Loops

If the optimization of masking is successful, the OWNED masks for the statements within a loop body are transformed into local loop bounds for the DO loop header. The parallelization of entire DO loops is indicated by masking the loop header as follows:

mask-> <do-stmt>

B.1.2 Communication Masks

VFCS uses additional masks to control the host-node communication and to determine a unique processor which is responsible for communicating (partially) replicated data. For these purposes the following masks are used:

MSP :
: The executing processor is the Master of all Segment Processors, an implementation dependent unique processor.

CSM (array) :
: The executing processor is the Communication Segment Master, an implementation dependent unique processor which owns a segment of the specified array. It is responsible for sending elements of that segment to other processors, and is selected by the system if a segment is owned by more than one processor (4.4).

LS (array) :
: The executing processor owns a local segment of the specified array (i.e. has a Local Segment).

B.2 Overlap Descriptions

This notation is used to describe the overlap area of distributed arrays as defined in Section 4.5.

{[dl1/du1],..[dln/dun]}

The above overlap description defines a rectilinear contiguous area around the local data segment of a processor. The first pair of values defines the extension in numbers of elements to the left and right of the local segment in the first dimension, and so on. A "*" indicates an entire dimension.
If the description is followed by "...", then more than one overlap is possible for the given statement, depending on run time conditions. This is possible only if dummy arrays are involved in the computation which leads to the overlap.

B.3 High Level Communication Statements - EXSR

EXchange Send Receive statements are used to indicate the exchange of data between node processors via explicit communication. After overlap analysis has been performed EXSR statements are extended by an overlap description.

EXSR <array_reference> <ovp_descr>

The above EXSR statement indicates the exchange of values of <array_reference> between processors depending on the overlap description given in <ovp_area_descr>.

B.4 Parallel Loops

Interactively specified `DO INDEPENDENT` loops are marked with the following notation:

DO INDEPENDENT
: The loop has successfully been marked as parallel. The information below is provided in addition.

DISTR ARRAYS <array-name>... :
: A list of the distributed arrays referenced in the parallel loop is given.

LOCAL SCALARS <variable-name>... :
: A list of the local scalar variables referenced within the parallel loop is given.

BLOCK :
: Indicates the selected work distribution.

CYCLIC :
: Indicates the selected work distribution.

ON OWNER (<array_element>) :
: Indicates the selected work distribution.

ON CONT (<array_element>) :
: Indicates the selected work distribution.

Appendix C

Menu Options

C.1 File Menu

The *File* menu is used to load and save programs, to select a project name, to load, save, or delete program versions, and to exit from VFCS.

C.1.1 Load Source Files

Selecting the *Load Source File* menu item results in the display of a file selector box. **All** the files that make up our program may be selected by clicking on each of the corresponding entries in the *Files* subwindow. After clicking the `OK` button, the files are loaded in alphabetical order.

Note: The system currently provides no support for separate compilation. Thus, if a program consists of more than one source file, all these files have to be selected.

C.1.2 Save Target Files

Selecting the *Save Target Files* menu item saves the actual version of the loaded program to file.

Case (i): If *Save Target Files* is selected before the *basic parallelization* has been applied, the actual program is save to the file *\$node.f*.

Case (ii): If *Save Target Files* is selected after the *basic parallelization* has been applied but before target code has been generated (i.e. *Generate Code* from the *Parallelization* menu), the actual program is saved to the files *\$host.f* and *\$node.f*. *\$host.f* contains the intermediate host program. *\$node.f* contains the intermediate SPMD node program.
Note: These files may contain intermediate language elements as described in Appendix B, and thus may not be processed further by other compilers.

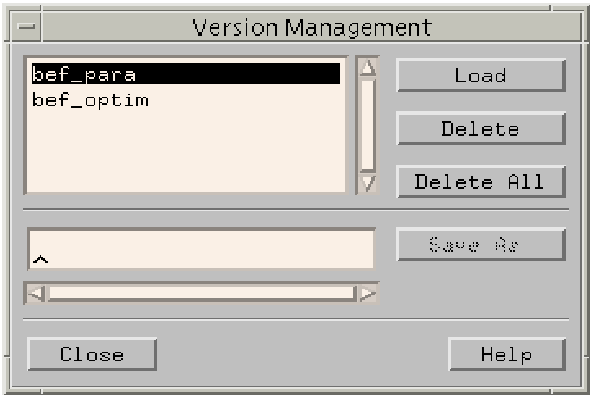
Case (iii): If *Save Target Files* is selected after *code generation*, the actual program is saved to the files *\$host.f* and *\$node.f*. *\$host.f* contains the host program and *\$node.f* contains the SPMD node program.

Note:
If the user has selected a *project name*, the "\$" in the host and node program filenames is substituted accordingly. If for example *proj1* has been selected as project name, the node program will be saved in *proj1_node.f*.

C.1.3 Version Management

The version management of VFCS enables the user to manage different program versions. This facility enables the user to save the current state of the transformation process. A program version consists of all the loaded program files together with all the analysis information. Program versions are not deleted upon exit from VFCS, and may be reloaded in a subsequent session.

Selecting the *Version Management* menu item results in the display of the *Version Management* window. The *version list* in the upper left part of this window, contains the existing program versions, if any. The text edit box, in the lower right part of this window, allows the user to enter a new name for saving the current version of the program.



- Load a Version
  Select a version by clicking on one of the entries in the *version list* and then push the *Load* button.
- Delete a Version
  Select a version by clicking on one of the entries in the *version list* and then push the *Delete* button.
- Delete All Versions
  Push the *Delete All* button.
- Save a Version
  Click at the text edit box, enter the name of the version you want to save, and push the *Save As* button.

Figure C.1: Version Management Window

Note: If you save a version, the actual program is not saved to a file as described in Section C.1.2. Only the internal representation of the program is saved.

C.1.4 Select Project Name

Selecting the *Select Project Name* results in the display of a window where you can select or define a project name, which is used as a prefix for the generated target files as described in Section 3.3.2.

C.1.5 Exit

Selecting the *Exit* menu item terminates the current session.

Note: The actual program and the corresponding analysis information will be lost, if it has not been saved previously. The program text can be saved by selecting *Save Target Files*, and the actual program version (including all the analysis information) by selecting *Version Management* as explained above.

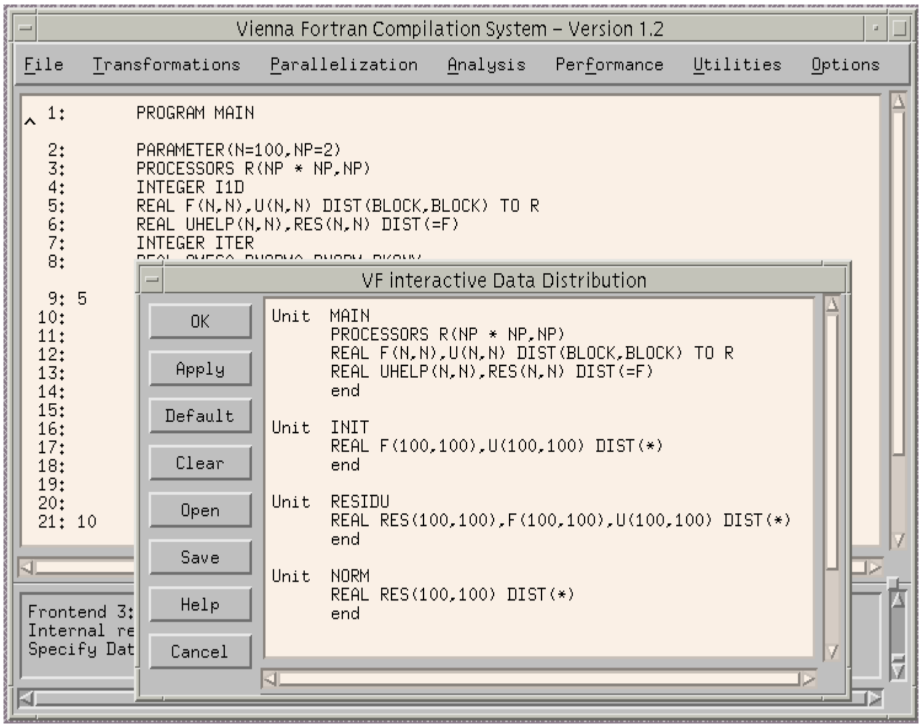
C.2 Transformation Menu

VFCS provides a number of standard transformations which can be selected manually from the *Transformation Menu*. Most of these transformations require the user to select a loop, a statement or a variable by clicking at the corresponding construct in the VFCS *Text Window*. In all cases where such a selection is requested from the user, the system displays appropriate message boxes. Furthermore, the system provides the user with additional information on the success or failure of transformations in the *Status Window*. For a description of the available transformations refer to Section 6.2.

C.3 Parallelization Menu

C.3.1 Data Distribution

Selecting the *Data Distribution* menu item results in the display of the *Interactive Data Distribution Window* (see Figure C.2) which enables you to specify or modify the distribution of arrays interactively.



OK Analyse input (if modified) and quit window if no error occurs.

Apply Analyse input (if modified), but do not quit window.

Default Reread all distributions of the program.

Clear Clear the window.

Open Read in distribution specifications from a file. This will display a file-selector box.

Save Save current distribution specifications to a file. This will display a file-selector box.

Cancel Quit the window without analyzing the input.

Figure C.2: Interactive Data Distribution Window

Whenever this window is activated, all the data distribution and alignment specifications of the program are displayed for each (node) program unit according to the following syntax rules.

C.3.1.1 Interactive Data Distribution Syntax

The syntax for specifying the data distribution interactively is in accordance with the syntax rules described in Appendix A. Note, however, that the interactive data distribution window, allows in some cases using several shortcuts. For example, the type and shape specification of arrays may be omitted. In the following only the top-level syntax rules are listed. For more details refer to Appendix A.

Note: It is not possible to intermix Vienna Fortran (77 or 90) distribution specifications and HPF directives.

Interactive Data Distribution Syntax

interactive-mapping	**is**	*vf-unit-mapping-list*
	or	*hpf-unit-mapping-list*

Vienna Fortran Interactive Data Distribution Syntax

vf-unit-mapping	**is**	`UNIT` *unit-name* [*vf-mapping*] ... `END` [*unit-name*]
vf-mapping	**is**	*vf77-mapping*
	or	*vf90-mapping*
vf77-mapping	**is**	*vf77-proc-decl-stmt*
	or	*vf77-reshape-decl-stmt*
	or	*vf77-mapping-decl*
vf77-reshape-decl-stmt	**is**	`RESHAPE` *array-declarator-list*
vf77-mapping-decl	**is**	*vf77-type-decl-stmt*
	or	*vf77-i-type-decl-stmt*
vf77-i-type-decl-stmt	**is**	[*type-spec*] *i-entity-decl-list declaration-annotation*
i-entity-decl-list	**is**	*array-name* [(*array-spec*)]
vf90-mapping	**is**	*vf90-proc-decl-stmt*
	or	*vf90-type-decl-stmt*

HPF Interactive Data Distribution Syntax

hpf-unit-mapping	**is**	`UNIT` *unit-name* [*hpf-mapping*] ... `END` [*unit-name*]
hpf-mapping	**is**	*processors-directive*
	is	*align-directive*
	is	*distribute-directive*
	is	*inherit-directive*

Note: Currently, alignment specifications can only be displayed but not modified within the *Interactive Data Distribution Window.* Thus, the system will report an error message if your program contains alignment specifications and you click at the *Apply* button.

C.3.2 Modify Work Distribution

Selecting the menu item *Modify Work Distribution* enables the user to modify the work distribution (i.e. the mapping of loop iterations to processors) of *INDEPENDENT* loops. The system offers the following options:

- Cyclic Work Distribution
- Block Work Distribution
- Work Distribution by Owner
- Work Distribution by Value
- Automatic Work Distribution

Note: *Modify Work Distribution* may not be selected before the *Basic Parallelization* transformation has been performed.

C.3.2.1 Cyclic Work Distribution

The `CYCLIC` work distribution maps the loop iterations in a round-robin fashion to the processors. Assume, for example, that the total number of loop iterations is given by L and the total number of available processors is P. Then, a particular processor p, $1 \le p \le P$ will execute all iterations $i, 1 \le i \le L$, for which $p = 1 + (i-1)\ mod\ P$ holds.

Note: *Cyclic* work distribution can only be specified interactively. The *ON CONT* clause is not accepted by the system if it is specified in the source code.

C.3.2.2 Block Work Distribution

Assume that L is the total number of iterations performed by the corresponding `DO INDEPENDENT` loop, and P is the total number of available processors. The total number of iterations L is divided into contiguous blocks, whose sizes differ by at most 1. Let $r := (L\ mod\ P)$, and $q := \lfloor \frac{L}{P} \rfloor$. The iteration space is divided into r blocks of size $q+1$ — which are mapped to the first r processors — and $P - r$ blocks of size q, mapped to the remaining processors.

Note: *Block* work distribution can only be specified interactively. The *ON CONT* clause is not accepted by the system if it is specified in the source code.

C.3.2.3 Work Distribution by Owner

Example C.1: Work Distribution by Owner

```
...
DO INDEPENDENT I=1,100 ON OWNER(X(I))
   A(X(I)) = B(Y(I))
END DO
...
```

In this example the i-th loop iteration will be performed by processor the processor which owns array element $X(i)$. Note that X must be a distributed array.

If *Work Distribution by Owner* is selected, the system asks the user to select a *DO INDEPENDENT* loop. After selection of a loop, the user has to select a statement inside this loop. Finally the system displays a window, where the user has to specify the name of a distributed array that is referenced within that statement. This array will be used in the *ON OWNER* clause.

C.3.2.4 Work Distribution by Value

If *Work Distribution by Value* is chosen, the system asks user to select a *DO INDEPENDENT* loop. After selection of a loop, the user has to specify the name of an distributed array, which will be used in the *ON CONT* clause.

Example C.2: Work Distribution by Value

```
...
DO INDEPENDENT I=1,100 ON CONT(X(I))
   A(X(I)) = B(Y(I))
END DO
...
```

In this example the i-th loop iteration will be performed by processor $X(i)$. Note that X must be an integer array and all values of X must correspond to a valid processor number.

Note: *Work Distribution by Value* can only be achieved interactively. The *ON CONT* clause is not accepted by the system if it is specified in the source code.

C.3.2.5 Automatic Work Distribution

If *Automatic Work Distribution* is chosen, the user is asked to select a *DO INDEPENDENT* loop. The system then searches for the first occurrence of a distributed array inside the body of the loop, say A, and uses this array according to the *Work Distribution by Owner*.

C.3.3 Basic Parallelization

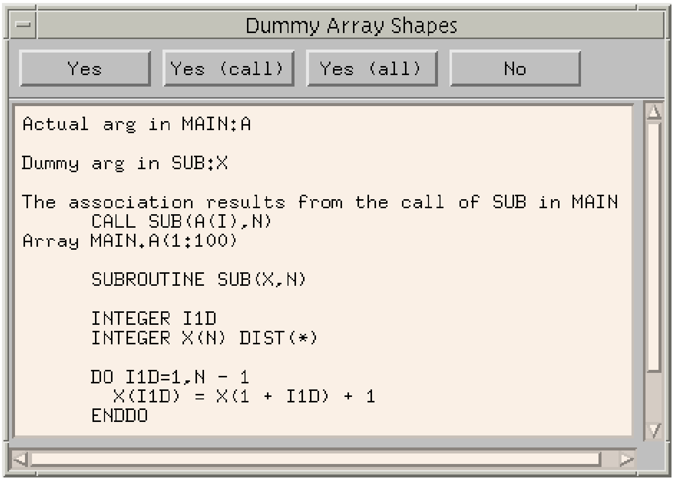
If the menu item *Basic Parallelization* is selected, the system performs automatically the following actions:

1. Splitting
2. Distribution analysis
3. Masking
4. Communication insertion

Usually the basic parallelization phase is done automatically. However during distribution analysis certain situations may be encountered where the user is asked to provide additional information to the system. For more details on these actions refer to Section 4.4.

C.3.3.1 Dummy Array Shape Window

The *Dummy Array Shape Window* may be displayed during distribution analysis to ask the user whether for a certain procedure call the dummy argument is always a subarray of the actual argument.



- `Yes`
  The dummy array is always a subarray of the actual array.
- `No`
  The dummy array is not a subarray of the actual array in at least one incarnation.
- `Yes(call)`
  For all other pairs of arrays in this call, the condition that the dummy array is a subarray of the actual array is fulfilled.
- `Yes(all)`
  The condition is fulfilled in all other situations occurring during the execution of this transformation.

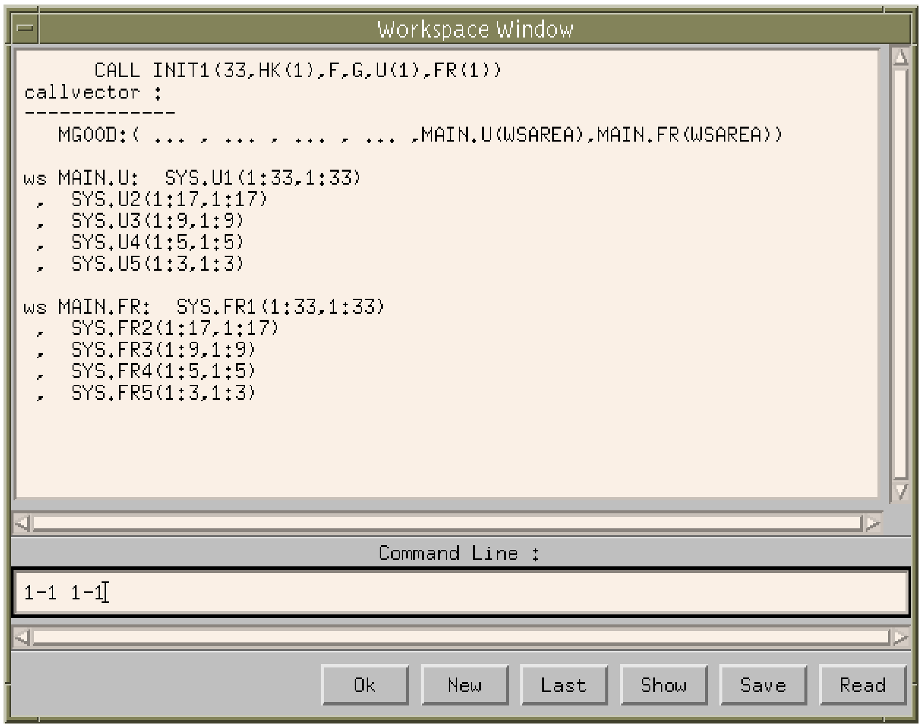
Figure C.3: Dummy Array Shape Window

The *Dummy Array Shape Window* displays the following information (see Figure C.3):

- the *CALL* statement
- the actual argument
- the dummy argument
- the source code of the called procedure

C.3.3.2 Workspace Window

If a program uses workspaces which are distributed as described in Section 2.2.7, then the system will display the *Workspace Window* during the basic parallelization phase to prompt the user for further information (see Figure C.4). For each incarnation of a `CALL` statement with a workspace array (element) as actual argument the system asks to specify which virtual arrays are associated with the corresponding dummy arrays.



Ok Accepts the input from the commandline. This has the same effect as pressing the *Return* key.

New Invalidates all combinations specified for that call.

Last Determines the last answer to be the current answer.

Show Reproduces the information window.

End Terminates the specification of combinations for this situation.

Save Starts the protocol for all supplied combinations. The system requires a file name.

Read Read specified combinations from a file. The system requires a file name.

Figure C.4: Workspace Window

The system presents in the *Workspace Window* the following information:

- the `CALL` statement, and the *call vector*
- the work spaces and their virtual arrays being accessed, and
- the last user-supplied answer to such a question (if syntactically possible in the current situation).

The user can select commands, using the command buttons, to determine all possible combinations of virtual arrays with dummy arrays in incarnations resulting from that call. Besides these commands the user may specify for each call statement the combinations of virtual arrays in the *Commandline*, using a special notation.

Specifying combinations in the commandline:

Before entering a specification at the command line of the *Workspace Window*, the user has to activate the commandline with a mouse click. A specification has to be made up of a single term for each dummy array accessing a virtual array of a workarray. A single term is either an integer constant **'n'** or a sequence **'n-m'** of constants. The specified constants have to be in the range between one and the number of virtual arrays in the corresponding workarray. Furthermore, if more than one sequence is given in a specification, the sequences must have the same number of elements. Each specification has to be terminated by an *end* command.

The combinations determined from such a specification are made up of the single constants and of the constants in the same positions of the sequences.

Furthermore, the information needed to perform distribution analysis for workspaces (i.e. the mapping of virtual arrays to dummy arrays) may be read from a file using the *Read* button.

C.3.4 Communication Optimization

Selecting the menu item *Communication Optimization* results in the activation of communication optimization transformations, with the goal of extracting communication statements from loops and combing small messages into larger messages. Communication optimization is done fully automatically.

C.3.5 Mask Optimization

Selecting the menu item *Mask Optimization* results in the activation of a transformation in order to adapt the bounds of `DO` loops that reference distributed arrays in such a way that each processor only executed those loop iterations that result in assignments to variables owned by it. Adaptation of loop bounds is indicated by using elements of the intermediate language as described in Appendix B. Mask optimization is done fully automatically.

C.3.6 Code Generation

Selecting the menu item *Code Generation* transforms the intermediate form of the parallelized and optimized program into a machine specific message passing Fortran 77 program. Intermediate communication statements are transformed into corresponding library calls. Masks of *DO* loops are transformed by modifying the loop bounds expressions. Declarations of distributed arrays are transformed such that each processor allocates the local part and an overlap area (if needed). Temporary variables needed for the communication between the processors are generated.

Code generation is done fully automatically.

Note: The target architecture, the communication library and the execution model may be chosen by the user by selecting the menu item *Code Generation Options* from the *Options* menu (See also Section 4.8).

C.4 Analysis Menu

C.4.1 Data Flow

Selecting the menu item *Data Flow* displays a submenu where the user may select one of the following options:

- *DU chains*
  This will result in showing the definition-use chains for all scalar variables defined in the selected statement.
- *Output chains*
  This will result in showing the output dependences for all scalar variables defined in the selected statement.
- *Anti chains*
  This will result in showing the anti dependences for all scalar variables used in the selected statement.
- *UD chains*
  This will result in showing the use-definition chains for all scalar variables which are used in the selected statement.
- *All chains*
  This will result in showing all the chains mentioned above for all scalar variables occurring in the selected statement.
- Use/Def Sets
  Selecting the menu item *Use/Def Sets* from the *Analysis* menu will display the variables written (marked as 'MAY_DEF' or 'MUST_DEF') or read (marked as 'USE') for every statement (identified by row number and statement type) for the actual program unit.

For more details please refer to Section 6.1.1

C.4.2 Data Dependencies

Selecting the menu item Data Dependencies displays a submenu where the user has the following options to choose one of the following dependence types:

- **All dependences**
- **True dependence**
- **Anti dependence**
- **Output dependence**

After having selected the dependence type, the user is asked to select a DO-loop or a statement within a DO-loop. In the first case the system will display all the dependences of the selected type and level (see below) for the selected loop. If a statement within a loop is selected, then all the dependences this statement is involved in are displayed.

Note: Dependence information is only provided for whole loops (or loop nests) or for single statements within a loop.

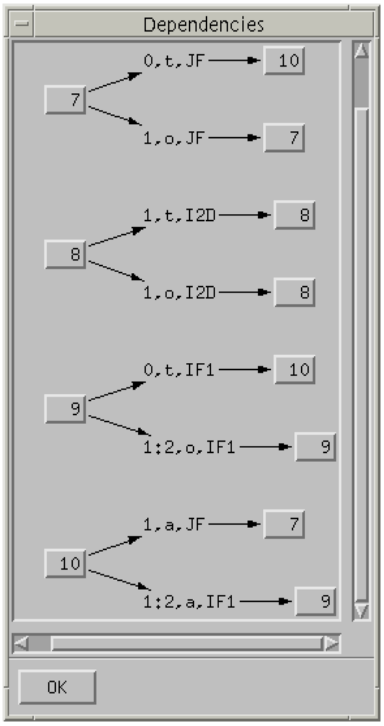
After selection of the dependence type and a loop or a statement, a window is displayed where the user has to select the dependence level as described in Figure C.5.

Selecting a Dependence Level

- **a**
  Clicking at the **a**-button will result in the display of all loop-carried dependencies (at all levels) as well as loop-independent dependencies.
- **0**
  Clicking at the **0**-button will result in the display of loop independent dependencies only.
- *Selection of the dependence level*
  The user may manually select a dependence level between 1 and 9 and the click the **OK** button. As a result only the dependences with the selected level (and of the selected type) will be displayed.

Figure C.5: Selecting a Dependence Level

The dependence information is visualized in a separate window (see Figure C.6) as a pseudo *dependence graph.*



Each edge of the dependence graph is represented in the following form:

row_nr1 —> (level1 [:level2], dep_type, var1, [var2]) —>row_nr2

- *row_nr1*
  identifies the statement (by displaying the row number) which is the source of the dependence
- *level1*
  specifies the dependence level of the displayed dependence
- *level2*
  can be shown only for scalar variables; if specified, *level1 : level2* gives the range of levels for which the displayed dependence exists
- *dep_type*
  specifies the dependence type of the displayed dependence
- *var1*
  denotes the variable of the statement in line *row_nr1* which causes the dependence
- *var2*
  denotes the variable of the statement in line em row_nr2 which causes the dependence; it is omitted if *var2* is the same as *var1*
- *row_nr2*
  identifies the statement (by displaying the row number) which is the sink of the dependence

Figure C.6: Data Dependence Window

C.4.3 Call Graph

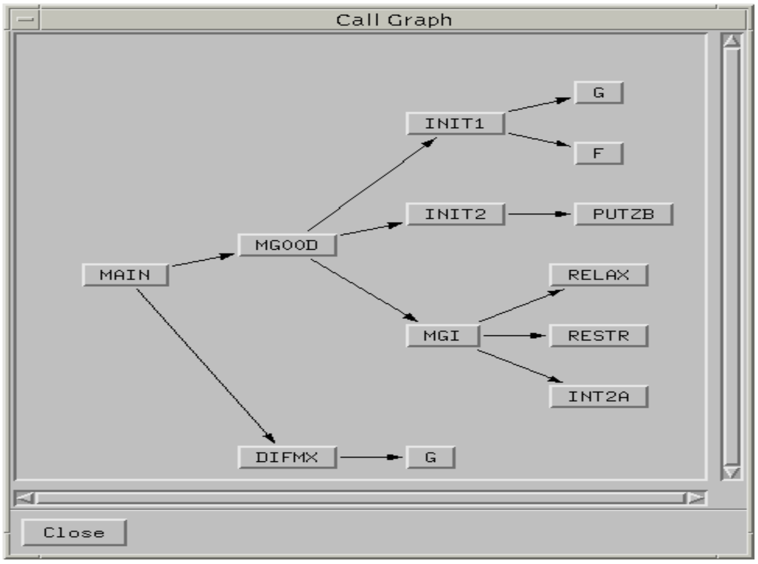


Figure C.7: Call Graph Window

Selecting the menu item *Call Graph* from the *Analysis* menu will display the call graph of the (host and node) program. The call graph can be used to load a program unit in the VFCS text window by clicking at the corresponding node in the graph.

C.4.4 Procedure Calls

Selecting the menu item *Procedure Calls* from the *Analysis* menu will display the formal parameters and all the *call vectors* of the actual unit. Each call vector is associated with a single incarnation of the actual program unit; it is defined as the vector of actual argument distributions that are associated with that particular incarnation of the procedure.

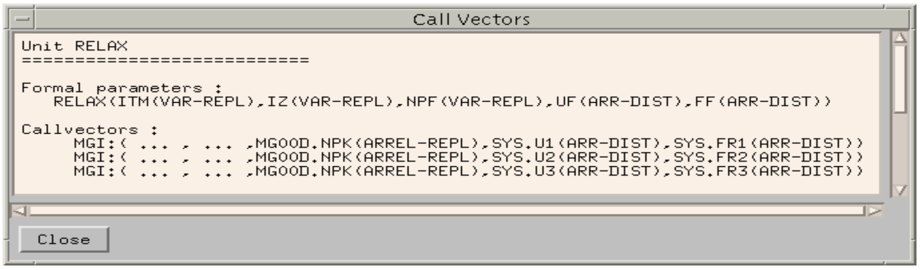


Figure C.8: Procedure Calls Window

Note: Information on procedure calls is not available before *Basic Parallelization.*

For each formal argument the following information is provided:

unit-name.name(*info*)

For each actual array argument in a call vector the following information is provided:

unit-name.array-name(*info*)

where *unit-name* and *array-name* is the name of the unit and the array to which the corresponding formal parameter is bound.

The *info* component consists of two parts separated by a "-" with the following meaning:

- The first part specifies whether the argument is a whole array or an array element.

  VAR indicates that the argument is a scalar varaiable
  ARR indicates that the argument is a whole array
  ARREL indicates that the argument is an array element

- The second part of the *info* component provides information on the distribution of the argument.

  REPL indicates that the corresponding argument is replicated (not distributed)
  DIST indicates that the corresponding argument has a standard distribution
  DIST? indicates that the corresponding argument has a non-standard distribution
  WORK indicates that the corresponding argument is a workspace array

Note: In a call vector only actual arguments that are arrays (or array elements) are shown.

C.4.5 Common Blocks

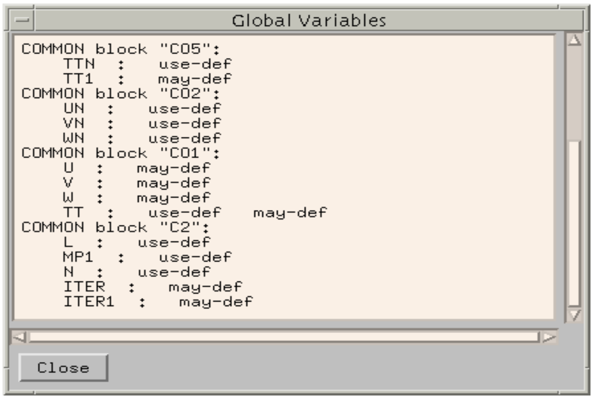


Figure C.9: Common Block Window

Selecting the menu item *Common Blocks* from the *Analysis* menu displays the dummy arguments (if any) and the variables of `COMMON` blocks accessed by the actual program unit. Every entry is marked as 'use_def' , 'may_def' or 'must_def', dependent on the first access (see Figure C.9).

C.4.6 Data Distribution

Selecting the menu item *Data Distribution* from the *Analysis* menu results in the display all distribution specifications of the entire program in a separate window.

C.4.7 Overlaps

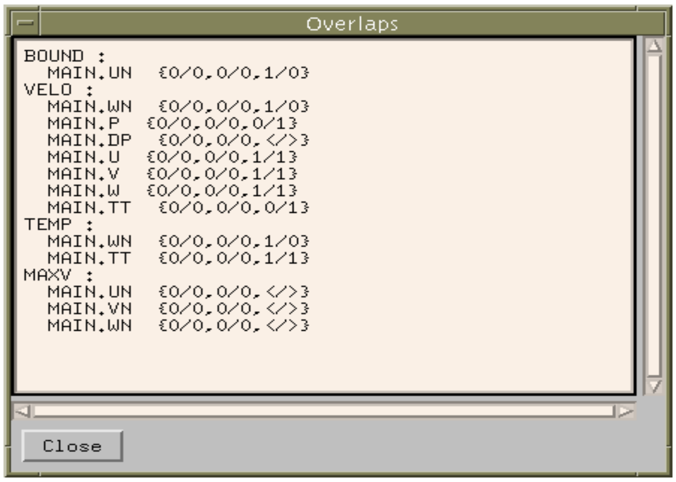


Figure C.10: Overlap Window

Selecting the menu item *Overlaps* from the *Analysis* menu displays a submenu with the following menu items.

- **Overlap In Unit**
  Selecting this submenu item displays all the array references that may have to be communicated together with the corresponding overlap description using the following form:

  line-number: *unit.array*{*overlap-descr*}

- **Overlap All Units**
  Selecting this submenu item displays all the overlap descriptions of the whole program. For each array only the maximum overlap description is displayed using the following format (see Figure C.10):

  unit.array{*overlap-descr*}

C.5 Utilities Menu

C.5.1 Units

Selecting the menu item *Units* displays a submenu with the following options:

- **Select Unit**
  Selecting this submenu item displays a window where all the node units and host units (if any) are listed. By clicking at a unit name and pressing the *OK* button (or by double clicking at a unit name), the corresponding unit is loaded in the VFCS main window and becomes the actual unit.

- **Save Unit**
  Selecting this submenu item displays a window where all units are listed. Clicking at a unit name and pressing the *OK* button (or by double clicking at a unit name) results in the display of a file-selector box where the user may specify the name of the file to which the selected unit will be saved.

- **View Unit**
  Selecting this submenu item displays a window where all units are listed. Clicking at a unit name and pressing the *OK* button (or by double clicking at a unit name) results in the display of the selected unit in an additional window.

- **Print Unit**
  Selecting this submenu item displays a window where all units are listed. Clicking at a unit name and pressing the *OK* button (or by double clicking at a unit name) causes the selected unit to be printed.

- **Delete All Units**
  Selecting this submenu deletes all units.

C.5.2 Goto Line

Selecting this menu item displays a window where the user may enter a line number. Clicking on the *OK* button positions the curser on the corresponding source code line.

C.5.3 Search

Selecting this menu item displays a window where the user may enter a pattern to be searched for. Clicking the *Find Next* button will highlight the next occurrence of the specified pattern. Clicking the *Show All* button will highlight all occurrences of the specified pattern.

C.5.4 History

Selecting this menu item redisplays all VFCS messages in the *Status Window*.

C.5.5 Protocol On/Off

Selecting the menu item *Protocol On* switches the protocol mode on. A file-selector box is displayed where the user has to specify the file name for the protocol file.

Selecting the menu item *Protocol Off* switches the protocol mode off and saves the current protocol file.

C.6 Options Menu

C.6.1 Code Generation

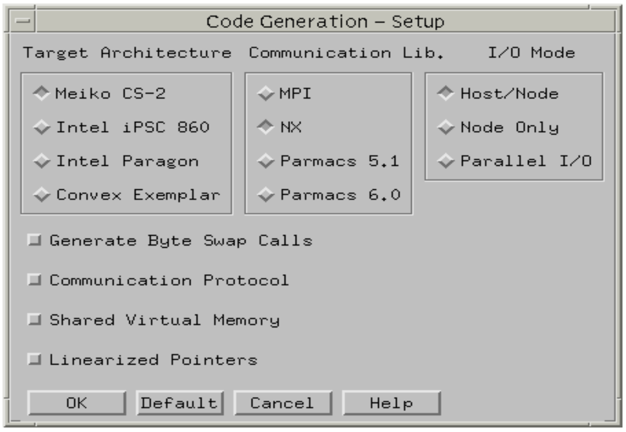


Figure C.11: Code Generation - Setup Window

Selecting the menu item *Code Generation* displays the *Code Generation - Setup Window* (see Figure refcodgensnap) where the user can select target architecture, communication library and communication model by clicking on radio buttons. Several options may be selected by using the appropriate checkboxes.

Special Options:

- Byte Swap
  If the *byte swap* switch is set, VFCS generates a call to particular byte swap routines after each host-node communication.
- Communication Protocol
  If the *communication protocol* switch is set, all communications in the program will be protocoled.
- Shared Virtual Memory
  Not yet available.
- Linearized Pointers
  Not yet available.

All parameters can be set multiple times during a VFCS session. It is also possible to select the current defaults by clicking on the 'default' pushbutton. When VFCS is left, the current parameter set becomes the default. (See also Section 4.8.)

C.6.2 Scanning Comments On/Off

If the menu item *Scanning Comments On* is selected, the system will read in the source file including comment lines.

If the menu item *Scanning Comments Off* is selected, the system will ignore all comment lines when source files are read.

Appendix D

Driver Directives

This chapter describes the driver directives which are used to construct driver files that control the actions taken by VFCS when executed in automatic mode. Usually, driver files are generated automatically by the system during an interactive parallelization session with the protocol mode turned on. Furthermore, driver files may be written by the user using the driver commands below. Note, however, that if driver files are generated manually, some transformations which require the selection of certain variables or loops may not be used.

For most of the available menu entries of VFCS corresponding driver command exist. These are described below.

D.1 File Commands

D.1.1 Selection of Source Files

Synopsis.

`add_source_file` *filename*

Arguments.

filename must be the name of a Fortran, Vienna Fortran, or HPF file.

Description.

The specified file is added to the list of selected files.

D.1.2 Loading Source Files

Synopsis.

`load_source_files`

Arguments.

None.

Description.

Loads source programs. Performs scanning, parsing, and code normalization, and initial program analysis.

Precondition.

At least one source file must have been selected either through a preceding *add_source_file* command or as a command line argument as described in Section 3.2.1

Note.

After the load source file command constant propagation should be performed by using the command `const_prop`.

D.1.3 Saving Target Files

Synopsis.

`save_final_code`

Arguments. None.

Description.

Saves the current state of the program. If this command is executed before the *basic parallelization* a single file containing the whole program is generated. If this command is executed after the *basic parallelization* a host program and a node program according to the naming conventions described in Section 3.3.2 is generated. Furthermore a make-file which can be used for generating object code and running the parallel program is generated.

D.1.4 Version Management

D.1.4.1 Loading Versions

Synopsis.

`get_version` *version-name*

Arguments.

version-name must be a valid filename.

Description. Loads a program version.

D.1.4.2 Saving Versions

Synopsis.

`set_version` *version-name*

Arguments.

version-name must be a valid filename.

Description.

Saves a program version.

D.1.4.3 Deleting Versions

Synopsis.

`del_version` *version-name*

Arguments.

version-name must be a valid filename.

Description.

Delete a program version.

D.2 Transformations

D.2.1 Induction Variable Substitution

Synopsis.

`induction_var`

Arguments.

None.

Description.

Performs induction variable substitution on the actual program unit.

D.2.2 Scalar Forward Substitution

Synopsis.

`scalar_fwd_all`

Arguments.

None.

Description.

Performs scalar forward substitution on the actual program unit.

D.2.3 Reduction Recognition

Synopsis.

`reduction`

Arguments.

None.

Description.

Performs reduction recognition on actual program unit.

D.2.4 Inline Expansion

Synopsis.

`inline_exp` *name*

Arguments.

name must designate a function or subroutine.

Description.

Performs inline expansion of a function reference or a subroutine call.

Note.

Must not be used for manually generated driver files.

D.2.5 Loop Distribution

Synopsis.

`loop_dist_all`

Arguments.

None.

Description.

Performs loop distribution within the actual program unit.

D.2.6 Loop Interchange

Synopsis.

`loop_interchg`

Arguments.

None.

Description.

Performs loop interchange for selected pair of loops.

Note.

Must not be used for manually generated driver files.

D.2.7 Loop Unrolling

Synopsis.

`loop_unroll` *factor*

Arguments.

factor designates the loop unroll factor.

Description.

Performs loop unrolling for selected loop.

Note.

Must not be used for manually generated driver files.

D.2.8 Loop Unroll and Jam

Synopsis.

`loop_unroll_jam` *factor*

Arguments.

factor designates the lopp unroll-jam factor.

Description.

Performs loop unroll and jam for selected loop.

Note.

Must not be used for manually generated driver files.

D.2.9 Mask Propagation

Synopsis.

`propagate_mask`

Arguments.

None.

Description.

Performs mask propagation on actual program unit.

Note.

Should be used after basic parallelization, but before communication optimization and mask optimization.

D.2.10 Strip Mining

Synopsis.

`loop_stripmine` *factor*

Arguments.

factor designates the blocking factor.

Description.

Performs strip mining for selected loop.

Note.

Must not be used for manually generated driver files.

D.2.11 Global Constant Propagation

Synopsis.

`const_prop`

Arguments. None.

Description.

Performs interprocedural constant propagation on whole program.

D.2.12 Expression Simplification

Synopsis.

`expr_simplify`

Arguments. None.

Description.

Performs expression simplification on actual unit.

D.2.13 Dead Code Elimination

Synopsis.

`dead_code`

Arguments. None.

Description.

Performs dead code elimination on whole program.

D.2.14 Vector Code Generation

Synopsis.

`gen_vec_code`

Arguments. None.

Description.

Performs vector code generation for all loops of the actual unit where this is semantically valid.

D.2.15 Transforming Vector Code into Loops

Synopsis.

`destroy_vec_code`

Arguments. None.

Description.

Transforms all vector statements of actual unit into corresponding `DO` loops.

D.3 Parallelization

D.3.1 Data Distribution

Synopsis.

`data_dist` [*filename*]

Arguments.

If *filename* is specified it designates a VF or HPF data distribution file.

Description.

Initiates data distribution.

Note.

A file with the data distribution specification has to be specified only in those cases where a sequential Fortran program (which does not contain any data distribution specifications) is processed.

D.3.2 Work Distribution

D.3.2.1 Marking Independent Loops

Synopsis.

`mark_indep_loop`

Arguments.

None.

Description.

Transforms a selected `DO`-loop into a `DO INDEPENDENT` loop.

Note.

Must not be used for manually generated driver files.

D.3.3 Basic Parallelization

Synopsis.

`start_par`

Arguments. None.

Description.

Generates initial parallel program.

D.3.4 Communication Optimization

Synopsis.

`opt_comm`

Arguments. None.

Description.

Performs communication optimization on whole program.

D.3.5 Mask Optimization

Synopsis.

`opt_mask`

Arguments. None.

Description.

Performs mask optimization on whole program by adapting loop bounds.

D.3.6 Code Generation

Synopsis.

`gen_code`

Arguments. None.

Description.

Generates parallel SPMD message passing code. Saves parallel program according to the conventions described in Section 3.3.2.

D.4 Units

D.4.1 Load Unit

Synopsis.

`select_unit` *unit-name*

Arguments.

unit-name must be the name of a program unit.

Note: The name of the main program is converted by the system to **`MAIN`**.

Description.

Loads the corresponding unit which becomes the actual unit.

D.4.2 Save Unit

Synopsis.

`save_unit` *unit-name file-name*

Arguments.

unit-name must be the name of a program unit. *file-name* must be a valid file-name.

Note: The name of the main program is converted by the system to **`MAIN`**.

Description.

Saves the program text of the actual unit to an external file.

Note: If the file *file-name* already exists, it is overwritten without any warning !

D.5 Positioning Commands

Note:

The positioning commands cannot be used for manually created driver files, since the system performs on each loaded program certain normalization transformations with the effect that the names of certain variables may be changed, certain language constructs may be transformed, and line numbers and labels may be modified. However, when a driver file is generated automatically using the protocol mode of VFCS, these commands are used in order to select certain program constructs and to apply transformations.

D.5.1 Selection of a Loop

Synopsis.

`search_loop` *loop-iteration-var*

Description.

Selects the first loop in the actual unit which has a loop iteration variable with the name *loop-iteration-var*.

Note.

Must not be used for manually generated driver files.

D.5.2 Selection of a Statement

Synopsis.

`search_stmt` *identifier*

Arguments.

identifier is either a label or a name.

Description.

If the argument is a label, the courser the statement of the actual unit with the corresponding label is selected. If the argument is an identifier, the first statement in the actual unit that references this *identifier* is selected.

Note.

Must not be used for manually generated driver files.

D.5.3 Selection of a Line Number

Synopsis.

`select_line` *number*.

Arguments.

number must be a valid line number with respect to the original source.

Note.

Must not be used for manually generated driver files.

D.6 Code Generation Options

The commands for setting target architecture specific code generation options described in the following may be specified optionally in a driver file. If a driver file does not contain such commands, the code generation options are read from the file *.besettings* located in the working directory. Thus, if VFCS is executed in automatic mode with a driver file that does not contain commands for setting code generation options, those of the last VFCS session will be used.

D.6.1 Byte Swapping

Synopsis.

`set_byte_swap` {on | off}

Arguments.

The argument must be *on* or *off*.

Description.

If byte swapping is turned *on* special byte-swapping routines are executed after each host-node communication. This is needed because some architectures use different byte orderings for host and node computer.

D.6.2 Selection of Communication Library

Synopsis.

`set_comm_lib` {MPI | NX | Parm_5.1 | Parm_6.0}

Arguments.

One of the above arguments must be specified.

Description.

Selects corresponding communication library.

D.6.3 Communication Protocol

Synopsis.

`set_comm_prot` {on | off}

Arguments.

The argument must be *on* or *off*.

Description.

If the communication protocol is turned *on* all communication statements of the program are protocoled.

Appendix E

Installing VFCS

This release of VFCS can be installed on most UNIX Workstations, including SUN SPARC, IBM, HP. Binary versions may be available for other workstations by arrangement.

This version requires the following:

- Operating System
  - SunOS 4.x, or
  - SunOS 5.x, or
  - AIX 3.2, or
  - HP-UX 9.05
- X Windows X11R5
- OSF Motif 1.1
- a minimum of 8 Megabytes disk storage; additional disk space is required for the parallelization of user code;

VFCS may be installed for a single user or for a number of users in a workstation network. It may generate code for the Intel iPSC/860 hypercube using the native message-passing library, or code for a machine on which PARMACS (Version 5.1 or 6.0) or MPI is available.

The system is available as a tar file via ftp by arrangement.

Installation

To install VFCS select a directory on which to install VFCS (e.g. `/usr/local`). This directory is further referred to as `<INSTALLDIR>`.

To extract the tar file enter the following commands.

```
cd <INSTALLDIR>
gunzip vfcs-1.2-solaris24.tar.gz
tar xvf vfcs-1.2-solaris24.tar
```

vfcs-1.2/			
			README
	bin/		
			VFCS binaries (vfcs, fe1, fe2, be)
	doc/		
		ps/	Postscript User's Guide
	lib/		
		autopar/	Default driver-file
		help/	Online help files (HTML)
		app-defaults/	Application default files
		ct_data_files/	P^3T data files
		make/	Makefile template
		spec/	internal error messages
		runtime/	Target architecture libraries
	examples/		
			Example codes

Figure E.1: VFCS Directory Structure.

This will create the files and subdirectories as shown in Figure E.1.

Environment Variables

Furthermore you have to make the following changes to your *.cshrc* or *.profile* file, depending on which shell you use. We assume in the following that *vfcsdir* corresponds to the directory `/usr/local`.

- Changes to *.cshrc* if you use *csh, tcsh*:

```
setenv VFCSROOT <INSTALLDIR>/vfcs-1.2          # environment variable needed by VFCS
setenv PATH $PATH:$VFCSROOT/bin               # add bin directory to your path
```

- Changes to *.profile* if you use *bash, sh, ksh, zsh*:

```
VFCSROOT=<INSTALLDIR>/vfcs-1.2                # environment variable needed by VFCS
export VFCSROOT
PATH=$PATH:$VFCSROOT/bin                      # add bin directory to your path
```

Runtime System

VFCS generates a *vfbuild* script for compiling the generated code on the target machine. There are two versions of the script template, one for *sh* (*bash, ksh, zsh*) and one for *csh* (*tcsh*). To specify this, a symbolic link has to be created as follows:

```
cd $VFCSROOT/lib/make
```

For *csh* use:

```
ln -s vfbuild.template.csh vfbuild.template
```

For *sh* use:

```
ln -s vfbuild.template.sh vfbuild.template
```

This directory also contains the Makefile template for building the generated code on the target machine. Look into this file to see if you have to make any adjustments (paths, compilers on the target machine).

Now, the runtime libraries have to be copied to the target machine. Choose your machine (IPSC, MEIKO, PARAGON). This will be referred as `<MACHINE>` now.

Create a *vfcs directory* on the target machine (like `/usr/vfcs` or `/opt/vfcs`), and copy

`<INSTALLDIR>/lib/runtime/<MACHINE>` and
`<INSTALLDIR>/lib/runtime/include`

recursively into the *vfcs directory* on the target machine.

Set `VFCSROOT` on the target machine to the *vfcs directory* there.