VIENNA FORTRAN Compilation System
Version 1.0
User's Guide

B.M. Chapman, S. Benkner, R. Blasko P. Brezany, M. Egg, T. Fahringer, H.M. Gerndt, J. Hulman, B. Knaus, P. Kutschera, H. Moritsch, A. Schwald, V. Sipkova, H.P. Zima
Department of Statistics and Computer Science,
University of Vienna, Brünner Strasse 72, A-1210 VIENNA, AUSTRIA

January, 1994

Contents

Chapter 1

Introduction

Vienna Fortran is a language extension of Fortran which provides facilities for the distribution of data across the processors of distributed-memory multiprocessing systems **(DMMPs)**. The **Vienna Fortran Compilation System (VFCS)** is an interactive system that performs a source-to-source translation from Vienna Fortran to explicitly parallel Message Passing Fortran, providing automatic parallelization as well as vectorization. VFCS is based upon the *data parallel model of computation* in connection with the *Single-Program-Multiple-Data (SPMD)* paradigm.

This User's Manual gives an overview of Vienna Fortran and describes the current version of VFCS, **VFCS 1.0**, in detail. Input languages for VFCS 1.0 are Vienna Fortran (VF) and FORTRAN 77 (F77). VFCS 1.0 generates code for the Intel IPSC-860 Hypercube, the SUPRENUM supercomputer and the Genesis-P machine. VFCS 1.0 can be used in an *interactive mode*. It offers a set of analysis services and a transformation catalog which allow the user to guide the system through the parallelization process. This mode is supported by a hierarchically structured menu system based on the X11-MOTIF user interface.
Alternatively, the system can be run in *batch mode*. A batch command language enables the creation of batch files which may be automatically applied to a Fortran program in this mode. Further, the sequence of transformations executed during an interactive parallelization session can be protocolled; the protocol can then be (automatically) executed in the batch mode.

This document is structured as follows:
In the rest of this chapter, we discuss DMMPs and their programming paradigms. The major design ideas of Vienna Fortran and VFCS are outlined and motivated, and related work is overviewed. Chapter 2 gives a brief description of Vienna Fortran. A complete language specification can be found in [41]. Chapter 3 explains the language of VFCS 1.0. This comprises a set of restrictions, modifications, and extensions of Vienna Fortran. Chapter 4 describes the basic compilation strategy, and Chapter 5 the user interface of VFCS 1.0. Chapter 6 describes the major transformations implemented in VFCS 1.0; these transformations are informally characterized and illustrated by examples. Chapter 7 discusses the batch facility, and Chapter 8 code generation. The appendices contain the data distribution language, the specification of the batch language, the VFCS 1.0 syntax, some example codes written in Vienna Fortran, and an installation guide.

1.1 Software for Distributed-Memory Multiprocessing Systems

The continued demand for increased computing power during the last decade has led to the development of computing systems in which a large number of processors are connected, so that their execution capabilities may be combined to solve a single problem.
In recent years, DMMPs (including Intel's hypercubes and the Paragon, the nCUBE, Meiko Computing Surface, and the CM5) have rapidly gained user acceptance. Other systems are under development or have been an-

nounced in recent months. These architectures are relatively inexpensive to build, and are potentially scalable to very large numbers of processors. Hence their share of the market is likely to increase in the near future.
The most important single difference between these 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.
A major difficulty with the current generation of distributed memory computing systems is that they generally lack programming tools for software development at a suitably high level. The user is forced to deal with all aspects of the distribution of data and work to the processors, and must control the program's execution at a very low level. 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. This has led to particularly slow software development cycles and, in consequence, high costs for software.
Thus much research activity is now concentrated on providing suitable programming tools for these architectures. One focus is on the provision of appropriate high-level language constructs to enable users to design programs in much the same way as they are accustomed to on a sequential machine. Several proposals (including Vienna Fortran) have been put forth in recent months for a set of language extensions to achieve this [6, 24, 31, 9], in particular (but not only) for Fortran, and current compiler research is aimed at implementing them.
Research in compiler technology has so far resulted in the development of a number of prototype systems, such as Kali [18], SUPERB [11, 38], and the MIMDizer [28]. In contrast to the current programming paradigm, these systems enable the user to write code using global data references, as on a shared memory machine, but require him or her to specify the distribution of the program's data. This data distribution is then used to guide the process of restructuring the code into an SPMD (Single Program Multiple Data) program for execution on the target DMMP. The compiler analyzes the source code, translating global data references into local and non-local data references based on the distributions specified by the user. The non-local references are satisfied by inserting appropriate message-passing statements in the generated code. Finally, the communication is optimized where possible, in particular by combining messages and by sending data at the earliest possible point in time.

Vienna Fortran is a machine-independent language extension to FORTRAN 77, which allows the user to write programs for DMMPs using global addresses. Since the performance of an SPMD program is profoundly influenced by the distribution of its data, most of the extensions are geared towards allowing the user to explicitly control such distribution of data. Vienna Fortran provides the flexibility and expressiveness needed to permit the specification of parallel algorithms and to carry out the complex task of optimization. Despite this fact, there are relatively few language extensions. A simple algorithm can be parallelized by the addition of just a few constructs which distribute the program's data across the machines.

1.2 Programming Distributed-Memory Systems Using Message Passing

The current generation of distributed-memory multiprocessors is particularly difficult to program: the time taken to adapt existing sequential codes and to develop new applications is prohibitive in comparison to conventional machines, including vector supercomputers. Further, the low level at which programs must be written is the source of both frequent errors and of particularly inflexible codes. Consider the brief example described below.
The Jacobi iterative procedure may be used to approximate the solution of a partial differential equation discretized on a grid. At each step, it updates the current approximation at a grid point by computing a weighted average of the values at the neighboring grid points. An excerpt from a Jacobi relaxation code for execution on a sequential machine is shown in Figure 1.1.
When this code is parallelized by hand, the programmer must distribute the program's work and data to the

```
C SEQUENTIAL CODE

      REAL UNEW(1:N,1:N), U(1:N,1:N), F(1:N,1:N)

      CALL INIT (U, F, N)

         ...

       DO 40  J = 2, N-1
         DO 40  I = 2, N-1

           UNEW(I,J) = 0.25 * (F(I,J) + U(I-1,J) + U(I+1,J) +
                               U(I,J-1) + U(I,J+1))

40     CONTINUE

          ...
```

Figure 1.1: Sequential Jacobi relaxation code.

processors which will execute it. One of the common approaches to do so makes use of the regularity of most numerical computations. This is the so-called *SPMD* (Single Program Multiple Data) or data parallel model of computation. With this method, the data arrays in the original program are each partitioned and mapped to the processors. This is known as *distributing* the arrays. The specification of the mapping of the elements of the arrays to the set of processors is called the *data distribution* of that program. A processor is then thought of as *owning* the data assigned to it; these data elements are stored in its local memory. Now the work is distributed according to the data distribution: computations which define the data elements owned by a processor are performed by it - this is known as the *owner computes* paradigm. The processors then execute essentially the same code in parallel, each on the data stored locally.

It is, however, unlikely that the code on one processor will run entirely without requiring data which is stored on another processor. Accesses to non-local data must be explicitly handled by the programmer, who has to insert communication constructs to send and receive data at the appropriate positions in the code. This is called *message passing*. The details of message passing 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. Moreover, for code which is explicitly parallel, debugging is a serious problem.

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.

The message-passing programming style requires that the communication statements be explicitly hardcoded into the program. But these statements are based upon the chosen data distribution, and as a result, the data distribution is also implicitly hardcoded. It will generally require a great deal of reprogramming if the user wants to try out different data distributions.

To illustrate this, we reproduce in Figure 1.2 the above section of code, rewritten to run on a set of P^2 processors using message passing code of the kind described. We have simplified matters by assuming that the processors have been organized into a two-dimensional array *PROC(P,P)* and that the processor array elements may be addressed for the purpose of exchanging data items: normally, a structure of this kind would have to be set up by the user first, and references would have to be converted to those provided by the environment. Further, we assume that the array sizes are multiples of P. Optimization of communication has been performed insomuch as messages have been extracted from the loops and organized into vectors for sending and receiving. When communication and computation are overlapped, as could be done here by carefully arranging the order in which local data is updated, the resulting code is considerably longer. Note that *SEND* and *RECEIVE* are assumed to be asynchronous.

In this version of the Jacobi relaxation, each processor has been assigned a square subblock of the original arrays. The programmer has declared local space of the appropriate size for each array on every processor. Array U has been declared in such a way that space is reserved not only for the local array elements, but also for those which are used in local computations, but are actually owned by other processors. This extra space surrounding the local elements is known as the overlap area. Values of *UNEW* on the local boundaries require elements of U stored non-locally for their computation. These must be received, and values from local boundaries must be sent to the processors which need them. Care is taken that the processors whose segments of U are on the original grid boundaries do not attempt to read from or send to non-existent processors.

The result of this low level style of programming is that the user spends a great deal of time organizing the storage and communication of data. In consequence, the time taken to produce a program is considerably longer than for comparable codes on shared-memory machines. Moreover, once written, the code is hard to modify or improve to run in some other way, even on the same machine. For example, if instead of dividing into square subblocks, the user wanted to experiment with blocking in only dimension, e.g., blocks of rows or columns, most of the code dealing with specification and communication would have to be modified. We use this example again in Chapter 4 to show how a Vienna Fortran version, consisting of the sequential program together with a data distribution, is parallelized by the Vienna Fortran Compilation System.

```
C PROCESSOR STRUCTURE PROC(P,P) IS ASSUMED

C CODE FOR PROCESSOR (P1,P2)

      PARAMETER( P = ... N = ...)
      PARAMETER( LEN = (N+P-1)/P )
C DECLARE LOCAL ARRAYS TOGETHER WITH OVERLAP AREA
C DATA OWNED LOCALLY IS U(1:LEN,1:LEN)
C AND SIMILARLY FOR UNEW AND F
      REAL U(0:LEN+1,0:LEN+1), UNEW(1:LEN,1:LEN), F(1:LEN,1:LEN)

      CALL LOCALINIT(U,F,LEN)
          ...

C SEND DATA TO OTHER PROCESSORS

      IF (P1.GT.1) SEND (U(1,1:LEN))   TO PROC(P1-1,P2)
      IF (P1.LT.P) SEND (U(LEN,1:LEN)) TO PROC(P1+1,P2)
      IF (P2.GT.1) SEND (U(1:LEN,1))   TO PROC(P1,P2-1)
      IF (P2.LT.P) SEND (U(1:LEN,LEN)) TO PROC(P1,P2+1)

C RECEIVE DATA FROM OTHER PROCESSORS, ASSIGN TO
C OVERLAP AREAS IN ARRAY U

      IF (P1.GT.1) RECEIVE U(0,1:LEN)     FROM  PROC(P1-1,P2)
      IF (P1.LT.P) RECEIVE U(LEN+1,1:LEN) FROM PROC(P1+1,P2)
      IF (P2.GT.1) RECEIVE U(1:LEN,0)     FROM PROC(P1,P2-1)
      IF (P2.LT.P) RECEIVE U(1:LEN,LEN+1) FROM PROC(P1,P2+1)

C COMPUTE NEW VALUES ON LOCAL DATA
      DO 40 I =  1, LEN
         DO 40 J = 1, LEN

            UNEW(I,J) = 0.25 * (F(I,J) + U(I-1, J) + U(I+1, J) +
                                  U(I, J-1) + U(I, J+1) )

40     CONTINUE
     ...
```

Figure 1.2: Jacobi relaxation code parallelized manually

Chapter 2

Vienna Fortran

This chapter presents a brief overview of **Vienna Fortran** [41], a machine-independent language extension to FORTRAN 77, which allows the user to write programs for distributed memory systems using global addresses. The Vienna Fortran language extension to Fortran 90 is described in [2]; its detailed specification is given in [41].
Note that we do **not** discuss the details of the language as currently implemented in VFCS 1.0 here – this will be done in Chapter 3. Note in particular, that there are interactive facilities for specifying data distributions which, in some cases, enable the user to provide distributions which are not easy to describe in the language or which enable the user to give the system enough information to construct a distribution. Also, the original methods in Vienna Fortran for distributing the work in a parallel loop have been extended in the implementation. Ongoing work includes extending the implementation subset of the language and providing the High Performance Fortran directives as an alternative notation.
After the discussion of the language features, the basic semantic model for Vienna Fortran will be specified in Section 2.2.

2.1 The Language Features

The Vienna Fortran language extensions provide the user with the following features:

- The **processors** which execute the program may be explicitly specified and referred to. It is possible to impose one or more structures on them.
- The **distributions** of arrays can be specified using annotations. These annotations may use processor structures introduced by the user.
  - Intrinsic functions are provided to specify the most common distributions.
  - Distributions may be defined indirectly via a map array.
  - Data may be replicated to all or a subset of processors.
  - The user may define new distribution functions.
- An array may be **aligned** with another array, providing an implicit distribution. Alignment functions may also be defined by the user.
- The distribution of arrays may be changed **dynamically**. However, a clear distinction is made between arrays which are statically distributed and those whose distribution may be changed at runtime.
- In **procedures**, dummy array arguments may

- inherit the distribution of the actual argument, or
- be explicitly distributed, possibly causing some data motion.

- A **forall** loop permits explicitly parallel loops to be written. Intrinsic reduction operations are provided, and others may be defined by the user. Loop iterations may be executed
  - on a specified processor, either by mapping them individually or in a block or cyclic fashion to processors,
  - where a particular data object is stored, or
  - as determined by the compiler.
- Arrays in **common blocks** may be distributed.
- **Allocatable** arrays may be used in much the same way as in Fortran 90. Array sections are permitted as actual arguments to procedures.
- **Assertions** about relationships between objects of the program may be inserted into the program.

We will in the following illustrate some of these features in more detail, using a set of simple code fragments. We use terminology and concepts from the definition of FORTRAN 77 (and, occasionally, Fortran 90) freely throughout.

2.1.1 The PROCESSORS statement

The user may declare and name one or more processor arrays by means of the **PROCESSORS** statement. The first such array is called the primary processor array; others are declared using the keyword **RESHAPE**. They refer to precisely the same set of processors, providing different views of it: a correspondence is established between any two processor arrays by column-major ordering of array elements as followed in FORTRAN 77. Expressions for the bounds of processor arrays may contain symbolic names, whose values are obtained from the environment at load time. Assertions may be used to impose restrictions on the values that can be assumed by these variables. This allows the program to be parameterized by the number of processors. This statement is optional in each program unit. For example:

```
PROCESSORS  MYP3(NP1, NP2, NP3)   RESHAPE   MYP2(NP1, NP2*NP3)
```

2.1.2 Processor References

Processor arrays may be referred to in their entirety by specifying the name only. Array section notation, as introduced in Fortran 90, is used to describe subsets of processor arrays; individual processors may be referenced by the usual array subscript notation. Dimensions of a processor array may be permuted.

2.1.3 Processor Intrinsics

The number of processors on which the program executes may be accessed by the intrinsic function *$NP* . A one dimensional processor array, **$P(1: $NP)**, is always implicitly declared and may be referred to. This is the default primary array if the processor statement is left out of a program unit. The index of an executing processor in **$P** is returned by the intrinsic function **$MY_PROC**.

2.1.4 Distribution Annotations

Distribution annotations may be appended to array declarations to specify direct and implicit distributions of the arrays to processors. Direct distributions consist of the keyword **DIST** together with a parenthesized *distribution expression*, and an optional **TO** clause. The **TO** clause specifies the set of processors to which the array(s) are distributed; if it is not present, the primary processor array is selected by default. A distribution expression consists of a list of distribution functions. There is either one function to describe the distribution of the entire array, which may have more than one dimension, or each function in the list distributes the corresponding array dimension to a dimension of the processor array. The elision symbol ":" is provided to indicate that an array dimension is not distributed. If there are fewer distributed dimensions in the data array than there are in the processor array, the array will be replicated to the remaining processor dimensions. Both intrinsic functions and user-defined functions may be used to specify the distribution of an array dimension.

```
REAL  A(L,N,M), B(M,M,M)    DIST ( BLOCK , CYCLIC , BLOCK )
REAL  C(1200)   DIST ( MYOWNFUNC )    TO    $P
```

The *BLOCK* intrinsic function distributes an array dimension to a processor dimension in evenly sized segments. The *CYCLIC* (or scatter) distribution maps elements of a dimension of the data array in a round-robin fashion to a dimension of the processor array. If a width is specified, then contiguous segments of that width are distributed in a round-robin manner.

There are two powerful variants of the *BLOCK* distribution which permit the user to distribute an array dimension to a processor dimension in uneven segments. These are the *S_BLOCK* and *B_BLOCK* distributions, for which the user supplies a list of segment lengths or upper bounds for the mapped segments, respectively. VFCS 1.0 provides additional block-based distribution methods (see Chapter 3).

A further option, known as **indirect** distribution, is to apply the intrinsic function *INDIRECT* to a *mapping array*, C. Let A denote the array to be distributed: for each index tuple $\mathbf{i}$, the element $A(\mathbf{i})$ is mapped to the processor whose number is given by the value of $C(\mathbf{i})$. This feature is useful in situations where the data structures of a program, and the associated access patterns, depend on input values and are determined at run time.

Another way to specify a distribution is to prescribe that the same distribution function be employed as that which was used to distribute a dimension of another array. For example,

```
REAL  D(100,100)   DIST (=A.1, =A.3)   TO  MYP2
```

will distribute D by BLOCK in both dimensions to the processor array *MYP2*. "*A.1*" refers to dimension 1 of array A while "*=A.1*" extracts the distribution of the first dimension of the array A. Note that both the extents of the array dimensions being distributed and the set of processors may differ from those of A.

Complex distributions may be constructed from the simpler ones: these are, in particular, necessary to provide efficient distributions for arrays used as work spaces, in which distinct parts of the array should often be mapped separately and in different ways to the processors.

Implicit distributions begin with the keyword **ALIGN** and require both the target array and a source array (so called because it is the source of the distribution). An element of the target array is distributed to the same processor as the specified element of the source array, which is determined by evaluating the expressions in the source array description for each valid subscript of the target array. Here, *II* and *JJ* are bound variables in the annotation, and range in value from 1 through 80.

```
INTEGER  IM(80,80)   ALIGN  IM(II,JJ)   WITH  D(JJ,II+10)
```

As is the case with direct distributions, the user may define functions to describe more complex alignments. By default, an array which is not explicitly distributed is replicated to all processors.

2.1.5 Dynamic Distributions and the DISTRIBUTE Statement

By default, the distribution of an array is static. Thus it does not change within the scope of the declaration to which the distribution has been appended. The keyword **DYNAMIC** is provided to declare an array distribution to be dynamic. This permits the array to be the target of a **DISTRIBUTE** statement. A dynamically distributed array may optionally be provided with an initial distribution in the manner described above for static distributions. A range of permissible distributions may be specified when the array is declared by giving the keyword **RANGE** and a set of explicit distributions. If this does not appear, the array may take on any permitted distribution with the appropriate dimensionality during execution of the program. Finally, the distribution of such an array may be dynamically connected to the distribution of another dynamically distributed array in a specified fixed manner. This is expressed by means of the **CONNECT** keyword. Thus, if the latter array is redistributed, then the connected array will automatically also be redistributed.

```
REAL F(200,200) DYNAMIC, RANGE((BLOCK,BLOCK),(CYCLIC(5),BLOCK))
```

An explicit distribution of an array is specified in a statement provided for this purpose. It begins with the keyword **DISTRIBUTE** and a list of the arrays which are to be distributed. Following the separator symbol "**::**", a direct, implicit or indirect distribution is specified using the same constructs as those for specifying static distributions.

```
        DISTRIBUTE  A,B :: ( CYCLIC (10))
```

2.1.6 Distribution Queries and The DCASE Construct

The **DCASE** construct enables the selection of a block of statements for execution depending on the actual distribution of one or more arrays. It is modeled after the CASE construct of Fortran 90. The keywords **SELECT DCASE** are followed by one or more arrays whose distribution functions are queried. The individual cases begin with the keyword **CASE** together with a distribution expression for each of the selected arrays. An asterisk, "*****", matches any distribution. The first case which satisfies the actual distributions of the selected arrays is chosen and its statements are executed. No more than one case may be chosen.

```
        SELECT    DCASE  (A, B)
            CASE  (( BLOCK ),( BLOCK ))
                CALL  BLOCKSUB(A,B,N,M)
            CASE  (( BLOCK ),( CYCLIC ))
                 ...
            CASE    DEFAULT
                 ...
        END    SELECT
```

The distributions of two arrays may be compared in a similar manner within an IF statement.

2.1.7 Allocatable Arrays

An array may be declared with the allocatable attribute as introduced in Fortran 90 by supplying the keyword **ALLOCATABLE**. The declaration defines only the rank of the array; the **ALLOCATE** statement is provided to allocate an instance of the array with specified bounds in each dimension. This instance is deallocated by means of the **DEALLOCATE** statement. An allocatable array may not be accessed if it is not currently allocated.

2.1.8 Common Blocks

Common blocks in which no data is explicitly distributed may be used as in FORTRAN 77. The common block storage sequence is defined for them. Individual arrays which occur in a named common block may also be explicitly and individually distributed just as other arrays are. However, they may not be allocatable and may not be dynamically distributed. Once storage space has been determined for a named common block, then it may not change during program execution.

2.1.9 Procedures

Dummy array arguments may be distributed in the same way as other arrays. If the distribution given differs from that of the actual argument, then redistribution will take place. If the actual argument is dynamically distributed, then it may be permanently modified in a procedure; if it is statically distributed, then the original distribution must be restored on procedure exit. This can always be enforced by the keyword **RESTORE**. While argument transmission is generally call by reference, there are situations in which arguments must copied. The user can suppress this by specifying a **NOCOPY**.
Dummy array arguments may also inherit the distribution of the actual argument: this is specified by using an "*" as the distribution expression:

```
CALL  EX(A,B(1:N,10),N,3)
      ...

SUBROUTINE  EX(X,Y,N,J)
REAL  X(N,N)   DIST (*)
REAL  Y(N)   DIST ( BLOCK )   TO  MYP2(1:N,J)
```

Array sections may be passed as arguments to subroutines using the syntax of Fortran 90.

2.1.10 The FORALL Loop

The **FORALL** loop enables the user to assert that the iterations of a loop are independent and can be executed in parallel. A precondition for the correctness of this loop is that a value written in one iteration is neither read nor written in any other iteration. There is an implicit synchronization at the beginning and end of such a loop. Private variables are permitted within forall loops; they are known only in the forall loop in which they are declared and each loop iteration has its own copy. The iterations of the loop may be assigned explicitly to processors if the user desires, or aggregates of them may be mapped in *BLOCK* or *CYCLIC* fashion to processors; alternatively, an iteration may be performed by the processor which owns a specified datum. Only tightly nested forall loops are permitted.

```
FORALL  I = 1, NP1*NP2*NP3   ON    $P (NOP(I))
INTEGER  K
       ...
END   FORALL
```

A reduction statement may be used within forall loops to perform such operations as global sums; the user may also define reduction functions for operations which are commutative and associative in the mathematical sense.

2.1.11 Input/Output

Files read/written by parallel programs may be stored in a distributed manner or on a single storage device. We provide a separate set of I/O operations to enable individual processor access to data stored across several devices.

2.2 The Semantic Model

2.2.1 The Data Space of a Program

Definition 2.2.1: data space

The **data space** *of a program is the set of data objects declared in the program. It is denoted by $\mathcal{A}$. $\mathcal{A}$ contains two classes of objects:* **scalars** *and* **arrays**.□ ■

A scalar object has a **rank** of 0. Arrays designate structured sets of scalars. They are characterized by an allocation status and have a rank ≥ 1, as defined below:

Definition 2.2.2: Array Allocation

1. *With respect to their storage allocation, arrays are classified into two groups:*
   - *A* **statically allocated array** *is allocated according to* FORTRAN *77 conventions. The* **allocation instance** *of such an array is defined to be the interval of time in which storage is allocated for the array during program execution.*
   - **Dynamically allocated arrays** *can be allocated and deallocated by explicit statements ALLOCATE and DEALLOCATE. The execution of ALLOCATE results in storage allocation for the array; DEALLOCATE releases the storage occupied by the last ALLOCATE statement executed for the array. The time interval between the execution of an ALLOCATE statement and the subsequent DEALLOCATE is called an* **allocation instance** *of the array. Between any two ALLOCATE statements applied to an array a DEALLOCATE must be executed; i.e., there is no nesting of allocations.*
2. *The* **allocation status** *of an array at a given time is* **allocated** *iff execution at that time is within an allocation instance for that array, otherwise it is* **deallocated**.

■

Definition 2.2.3: Index Domains

1. *An* **index domain of rank (dimension)** *n is any set $\mathbf{I}$ that can be represented in the form $\mathbf{I}=\mathbf{X}_{i=1}^{n} D_i$, where $n \geq 1$ and for all $i, 1 \leq i \leq n$, D_i is a nonempty, linearly ordered set of integer numbers. $\mathbf{I}$ is called a* **standard index domain** *iff each D_i is of the form $D_i = [l_i : u_i]$, where $l_i \leq u_i$ and $[l_i : u_i]$ denotes the sequence of numbers $(l_i, l_i+1, \ldots, u_i)$. l_i and u_i are then respectively called the* **lower** *and* **upper bound** *of dimension i.*

   In the following, let $\mathbf{I}$ denote an index domain of rank n, and i an integer number with $1 \leq i \leq n$.
2. *The projection of $\mathbf{I}$ to its i-th component, D_i, is denoted by $\mathbf{I}_i$.*
3. *$\mid D_i \mid$ is the* **extent** *of dimension i.*
4. *The* **shape** *of $\mathbf{I}$ is defined by $shape(\mathbf{I}) := (\mid D_1 \mid, \ldots, \mid D_n \mid)$.*

■

Definition 2.2.4: Array index domains

Assume that $A \in \mathcal{A}$ is an arbitrary declared array.

1. *If the allocation status of A is* **allocated**, *then:*
   (a) *A is associated with a standard index domain, $\mathbf{I}^A$. All attributes of $\mathbf{I}^A$, such as rank and shape, are applied to A with the same meaning as specified in Definition 3.*[1]
   (b) *$\mathcal{E}^A$ is the* **set of elements** *of A. The elements are scalar objects.*
   (c) *$index^A : \mathcal{E}^A \rightarrow \mathbf{I}^A$ is a function establishing a one-to-one correspondence between $\mathcal{E}^A$ and $\mathbf{I}^A$. For every array element $e \in \mathcal{E}^A$, $index^A(e)$ is called the* **index** *of e.*

[1]Whenever A is implied by the context, the superscript may be omitted. Analogous conventions hold for all similar cases.

2. If the allocation status of A is **deallocated**, *then only the rank of A is known. Neither an index domain nor a set of elements is associated with A.*

■

Definition 2.2.5:

Consider an arbitrary point t in time during the execution of the program. Then:

$\mathcal{E} := \{e \mid e \in \mathcal{E}^A$ *for some* $A \in \mathcal{A}$ *such that the allocation status of A is* **allocated**.$\}$

$\mathcal{E}$ is called the set of all array elements in the data space at time t.$\square$

■

The bounds, extents, and sizes of allocated arrays can be accessed in the program by the intrinsic functions *LBOUND, UBOUND*, and *SIZE*.
Until now we have discussed arrays in the data space, which are introduced by an array declaration. Arrays may also be computed at run time by forming a *section* of a declared array. Such arrays can be characterized in the same way as allocated arrays in the above definition; their index domain, however, may be nonstandard. In the following, if nothing is said to the contrary, the allocation status of an array will always be assumed to be **allocated**.

2.2.1.1 Processors

The **set of processors**, P, is represented in a program by one or more **processor arrays**, which provide a means of naming and accessing individual processors and subsets of processors. We will use the notational conventions introduced for arrays above; in particular, for a processor array R, $\mathbf{I}^R$ denotes the associated standard index domain, and $index^R : P \rightarrow \mathbf{I}^R$ the function mapping processors to their index.
Any two processors in P communicate by exchanging messages. Our model abstracts from the machine topology, such as grid, torus, or hypercube and the related message passing mechanisms, and thus does not reflect different processor "distances".

2.2.1.2 Distributions

A **distribution** of an **allocated** array maps each array element to one or more processors, which become the **owners** of the element, and, in this capacity, store the element in their local memory. We model distributions by functions between the associated index domains.

Definition 2.2.6: Index Mappings

Let $\mathbf{I}, \mathbf{J}$ denote two index domains. An **index mapping** *from $\mathbf{I}$ to $\mathbf{J}$ is a total function $\iota : \mathbf{I} \rightarrow \mathcal{P}(\mathbf{J}) - \{\phi\}$, where $\mathcal{P}(\mathbf{J})$ denotes the power set of $\mathbf{J}$.*

■

Definition 2.2.7: Distributions

1. Let $A \in \mathcal{A}$ denote an **allocated** *array, and assume that R is a processor array. An index mapping δ^A_R from $\mathbf{I}^A$ to $\mathbf{I}^R$ is called a* **distribution** *for A with respect to R.*

2. *Assume that δ_R^A is a distribution. Then $\hat{\delta}_R^A$ is the associated* **element-based distribution** *that maps elements of A to processors in P. It is defined as follows:*
   (a) *$\hat{\delta}_R^A : \mathcal{E}^A \rightarrow \mathcal{P}(P) - \{\phi\}$ is a total function.*
   (b) *For each $e \in \mathcal{E}^A$: $\hat{\delta}_R^A(e) = \{p \in P \mid index^R(p) \in \delta_R^A(index^A(e))\}$.* □

■

Note that δ_R^A uniquely determines $\hat{\delta}_R^A$, and vice versa.
Until now, we have only considered distributions of individual arrays. By generalizing the functions $\hat{\delta}_R^A$ in such a way that they combine the distributions of all arrays in the data space we obtain a **distribution state** as defined below. In most of the following, we will assume a fixed processor array R to be implicitly given, and we thus omit R in the notation except where indicated otherwise.
Let t denote an arbitrary point in time during the execution of a program, and assume that $A \in \mathcal{A}$ is an **allocated** array. At time t, A may or may not be associated with a distribution. If it is, the distribution is uniquely determined, if it is not, we say that the distribution of A at time t is **undefined**.

Definition 2.2.8: distribution state

The **distribution state at time** *t, $\hat{\delta} : \mathcal{E} \rightarrow \mathcal{P}(P) - \{\phi\}$ is given as follows: For each e such that $e \in \mathcal{E}^A$:*

- *If A is associated with a distribution $\hat{\delta}_R^A$ at time t, then $\hat{\delta}(e) := \hat{\delta}_R^A(e)$.*
- *If the distribution of A at time t is undefined, then $\hat{\delta}(e)$ is undefined.*

■

Definition 2.2.9:

Assume that a distribution state $\hat{\delta}$ is given. Let the total function $\lambda : P \rightarrow \mathcal{P}(\mathcal{E})$ be defined as follows: For each $p \in P, \lambda(p) = \{e \in \mathcal{E} \mid p \in \hat{\delta}(e)\}$. $\lambda(p)$ is the set of **local variables** *of p; these variables are also said to be* **owned** *by p.* □ ■

We finish this section with a few remarks.
In the context of distributions, we have not discussed scalars up to now. The specification and use of scalar objects in Vienna Fortran may be exactly as in sequential programs, without any additional specification. In this case, we assume **replication** by allocating space on each processor, thus avoiding all communication. On the other hand, Vienna Fortran provides a means to explicitly specify the owner(s) of scalar objects.
We can easily include scalars into our model by considering them (for this purpose only) as arrays with exactly one element. Such objects are always **allocated** and always have a well-defined distribution associated with them. The concept of the element-based distribution, the meaning of the set $\mathcal{E}$, and of the distribution state at a given time must then be suitably extended. We will use this generalization in just a few places in this document.
There are two ways in which allocated arrays can be associated with distributions. Any such association is valid within the program unit in which the declaration of the array occurs:

- The distribution type of a **statically distributed array** is evaluated at the time its declaration is evaluated. It is associated with all instances of the array.
- The distribution type of a **dynamically distributed array** may be different for different allocation instances, and, moreover, may change within any given allocation instance. Such a change can be caused

by the execution of a distribute statement or by a procedure call. As long as no explicit association between an array and a distribution is established in an allocation instance, the distribution of the array remains undefined.

Array elements can be legally accessed (read or written) only if the program executes within an allocation instance for that array **and** a well–defined distribution has been associated with the array.

2.2.1.3 Alignment

If A, B are different arrays used in a common context (for example, within one assignment statement), then the relationship of δ^A and δ^B may determine the amount of communication to be generated. If these functions are in an invariant relationship with each other during a certain phase of program execution, then we say that A and B are **aligned** during that period. For example, if we know that $\mathbf{I}^A = \mathbf{I}^B$ and $\delta^A = \delta^B$, then no communication needs to be generated for the assignment $A(I) = B(I) + 1$, if it is executed in the process which owns $A(I)$ and $B(I)$. We now describe this concept more precisely.

Definition 2.2.10:

Let $A, B \in \mathcal{A}$ denote arbitrary arrays. An index mapping α from $\mathbf{I}^A$ to $\mathbf{I}^B$ is called an **alignment** *for* **target array** *A with respect to* **source array** *B.* □ ■

If A, B, δ^B, and α are given as above, then δ^A can be computed as shown below:

Definition 2.2.11: Construction of a distribution

Let $A, B \in \mathcal{A}$, δ^B and an alignment function $\alpha : \mathbf{I}^A \rightarrow \mathcal{P}(\mathbf{I}^B) - \{\phi\}$ be given. Then we determine δ^A as follows: For each $\mathbf{i} \in \mathbf{I}^A$:

$$\delta^A(\mathbf{i}) := \bigcup_{\mathbf{j} \in \alpha(\mathbf{i})} \delta^B(\mathbf{j})$$

□ ■

Chapter 3

VFCS 1.0

In this chapter, we discuss the language features implemented in the current version of VFCS. Although the system restricts the user to a subset of Vienna Fortran, many additional features are available via the *interactive data distribution language*. Thus it is possible to write a program in FORTRAN 77 or Vienna Fortran and then use the interactive data distribution facilities, for example, to distribute (or redistribute) program data. The interactively specified distributions may be written to file (or the user may write a file to describe them), so that this information, too, is retained.
In Section 3.1 below, we describe the subset of Vienna Fortran implemented by listing those features which are *not* included. The syntax for the implemented features is to be found in Appendix C. In some cases, we indicate how they are nevertheless available in the system.
In the following section, 3.2, we discuss the interactive parallelization options. The main part of this section is concerned with the methods provided in the interactive data distribution language for describing data distributions, since further information on the interactive facilities for distributing FORALL loops is to be found in Section 6.6. We briefly describe how the interactive distribution language may be used to construct files containing the pertinent distribution information.
This section describes the basic constructs and strategies available and provides examples illustrating some of the most important features of the interactive language. The full functionality of the language is not included here, however; a precise description of the language is to be found in Appendix A. Recall that these options are available for both FORTRAN 77 and Vienna Fortran programs.
We conclude with a discussion of some general restrictions on the ways in which arrays may be distributed.

3.1 Parallelization of Vienna Fortran Programs

For programs written in Vienna Fortran the data distribution is specified via distribution annotations, which are appended to array declarations (2). The current version of VFCS implements a subset of Vienna Fortran, which is described below. We list the features which are not available. Note that a Vienna Fortran program may make use of all facilities described for the parallelization of FORTRAN 77 codes. The restrictions which may apply to arrays which appear in COMMON blocks or EQUIVALENCE statements hold for Vienna Fortran programs, and are discussed separately.

3.1.1 Constraints on the Usage of Vienna Fortran

The VFCS 1.0 subset implementation is restricted to static features, i.e. dynamically distributed arrays are not supported. Furthermore, at least one processor array must be defined in the main program unit. The size of processor arrays must be a compile time constant. Processor array declarations are not allowed inside procedures (i.e. all program units use the same processor array).

For the specification of distributions only the distribution functions BLOCK, B_BLOCK, S_BLOCK, ":", and distribution extraction are provided.
Cyclic distributions and distribution by alignment are not implemented.
New executable statements provided in addition to FORTRAN 77 are the FORALL loop, which is restricted to one-dimensional arrays, and the REDUCE statement, which may be used only inside FORALL loops. A precise discussion of the FORALL loop as implemented is to be found in 6.6.
User defined distribution functions, user defined alignment functions, the DCASE construct, concurrent I/O statements, and allocatable arrays are currently not in the subset.
Dummy arrays in procedures may be given an inherited distribution annotation only. If no distribution annotation is present for a dummy array argument, its distribution is assumed to be inherited by default.

3.1.2 Features of Vienna Fortran Not Available

This list is based on the Vienna Fortran language specification:

1. **Basic Elements:** (cf. VF language specification, page 17)
   (a) Assertions are not implemented. The system will, however, query the user to obtain information critical for the transformation process.
   (b) Array sections may be used in processor references only (i.e. array-sections must not be passed to procedures !) The system assumes standard FORTRAN 77 sequence association at procedure boundaries. If a dummy argument is an adjustable array, the system may query the user about the nature of the actual arguments.
   (c) "Assumed shape array declarators" are not allowed.
   (d) The *distribute* statement is not implemented.
   (e) The *allocate* and *deallocate* statements are not implemented.
   (f) The *dcase* construct is not implemented.
   (g) Concurrent I/O is not implemented.
2. **Processor Declarations:** (cf. VF language specification, page 20,21)
   (a) At least one processors statement must be specified for VF programs (in contrast to FORTRAN 77 programs).
   (b) The number of processors must be a compile-time constant.
   (c) Processor declarations must not appear inside procedures.
   (d) The implicit processor array \$P as defined by VF is not available (– it is currently available only for FORTRAN 77 programs).
   (e) The intrinsic function \$NP is not available.
   (f) The intrinsic function \$MY_PROC is not available for the user.
3. **Processor References**: (cf. VF language specification, page 22)
   These are fully implemented.
4. **Distribution Expressions**: (cf. VF language specification, page 23 ff)
   - **Implemented**:
     (a) Simple disribution expressions,
     (b) Composite distribution expressions,

(c) Distribution extraction.

- **Not Implemented**:

  (a) Distribution types,

  (b) User defined distribution functions. A number of user defined distribution functions are available for workspace arrays in the interactive data distribution language.

- **Restrictions**: only ":" and the intrinsic distribution functions are permitted.

5. **Static Array Annotations**: (cf. VF language specification, page 33)

   - **Not implemented**: alignment specification. A method for specifying alignments is available in the interactive data distribution language.
   - **Restrictions**: dummy arguments of procedures may have inherited distribution only.

6. **Dynamic Array Annotations**: not implemented.

7. **FORALL Loops**:

   The following restricted syntax is implemented:

```
forall-loop     ->  FORALL label forall-control
                                 forall-body
                    label CONTINUE

forall-control  ->  var_name = iexpr,iexpr [,iexpr] [work-dist-spec]


work-dist-spec  ->  ON OWNER (array_element)
                ->  ON proc-array-element
```

8. **Reduction Statement**: User defined reduction operators are not available in this version.

9. **Assertions**: not implemented.

10. **VF Intrinsic Functions**: only BLOCK, B_BLOCK, S_BLOCK are implemented.

11. **Concurrent I/O**: not implemented

3.2 Parallelization of FORTRAN 77 Programs

The parallelization of FORTRAN 77 programs (or the interactive specification or modification of distributions for Vienna Fortran programs) is achieved within VFCS by selecting the appropriate item of the *Start* menu and then entering the distributions interactively, by means of the *interactive data distribution language*. This data distribution language permits the specification of arbitrary block distributions and provides a concept which is similar to the Vienna Fortran processor arrays (see below). Alternatively, this language may be used to create a file describing the data distribution: the file is then read in by means of the same menu item. The syntax of the distribution language is contained in Appendix A.
All distributions that can be described with the implemented subset of Vienna Fortran can also be expressed in the data distribution language. Furthermore, the data distribution language provides additional features which currently are not supported by Vienna Fortran (in particular, additional methods for distributing a dimension of an array, and the workspace concept). Hence this is currently a much more powerful distribution method.

A distribution for an array is specified by partitioning the array into segments, and mapping the segments to the processes. A segment is a contiguous section of the array.
The partition is specified by subdividing the index set of each dimension of the array into contiguous parts. This can be specified using one of the following strategies:

- Default interactive strategy
- Vienna Fortran strategy
- Equal interior blocks strategy
- General block strategy

Common to all of these strategies is that the user determines the number of segments an array will be divided into. They vary in the method used to determine the length of each part of a dimension. Not all of the dimensions of an array must be distributed. The interactive language provides a powerful means for mapping the segments of an array thus specified to the set of processors: the segments are mapped to processor arrays in a flexible manner.

We describe the strategies offered in 3.2.1 below. This is followed by examples of their use, and a discussion of the workspace concept, for which an interactive form of user-defined distribution is available.

VFCS also provides interactive means for specifying FORALL loops and modifying their distribution, i.e. the method by which the loop iterations are distributed to processors for their execution. We describe these in 3.2.4.
This is followed by a brief description of the files which may be used to describe a data distribution.

3.2.1 Interactive Data Distribution

In the data distribution language, the mapping of segments of arrays onto processors is specified by declaring and using one or more *Processor Array Structures (PAS)*. Each segment is mapped to an element of a *PAS*. It is possible to map an element of a *PAS* to more than one processor of another defined *PAS*: this enables replication of array segments to a set of processors (see the example in 3.2.2.2. At most one segment of an array can be assigned to a single processor of a *PAS*.

Hence a *PAS* is a multi-dimensional array of logical processors, whose elements each identify one or more (!) processors. In VFCS 1.0 there is one predefined *PAS*, the linear processor array:

```
SP(1:$NP)
```

Each address SP(I) identifies one processor. The user can specify a new *PAS* by defining its structure and the mapping of its processors onto the processors of an existing *PAS*. One-to-many mappings are allowed, but many-to-one mappings are not permitted. The notation provided is illustrated in the following example.

Example:

```
pas p1(4) with (i=1,4 p1(i) -> sp(4+i) )
```

The *PAS* p1 is defined with four processors, p1(1), p1(2), p1(3) and p1(4). These processors are equivalenced with processors in the linear processor array (i.e. provide an alternative means to specify a set of processors of SP). They refer to processors of SP in the following way:

```
p1(1) -> sp(5)
p1(2) -> sp(6)
p1(3) -> sp(7)
p1(4) -> sp(8)
```

An array mapped to p1 is thus mapped to the processors SP(5:8).

Note that this provides a method for mapping arrays to a variety of subsets of the executing processors. This includes, for example, mapping the elements to the processors on the diagonal of a two-dimensional processor array structure (cf. the *Matrix* demo program delivered with the system).

We now outline the strategies for determining the array segments.

3.2.1.1 Default Interactive Strategy

In the default strategy, the user simply specifies the number of parts an array dimension is to be subdivided into, and the system computes the lengths of the parts according to the following algorithm: If the index set cannot be divided exactly by the specified number, the remainder is distributed equally among the parts in such a manner that the lengths of different parts differ by at most one and the shorter parts are placed on the left and right ends of the dimension.

Example: Let A(100) be an array.

```
part A(7)
```

determines a partition of A according to the default strategy. The segments are:

```
A(1:14)   A(15:28)  A(29:42)   A(43:57)
A(58:72)  A(73:86)  A(87:100)
```

The segment lengths are 14, 14, 14, 15, 15, 14, 14.

3.2.1.2 Vienna Fortran Strategy

The Vienna Fortran distribution strategy also divides the array dimension into contiguous blocks, whose sizes differ at most by one. If the number of array elements N in a distributed dimension is not exactly divisible by the number of processors P (in that dimension) then the array dimension is partitioned into r parts of size N/NP+1 which are mapped to the first r processors (r = N mod P), and NP-r parts of size N/NP which are mapped to the remaining processors.

Example: Let A(100) be an array

```
part A(7vf)
```

determines a partition of A according to the Vienna Fortran strategy. The segments are:

```
A(1:15)   A(16:30)  A(31:44)   A(45:58)
A(59:72)  A(73:86)  A(87:100)
```

The segment lengths are 15, 15, 14, 14, 14, 14, 14.

3.2.1.3 Equal Interior Blocks Strategy

This strategy splits the index set into a number of parts in such a way that all parts have the same length except for the leftmost part and the rightmost part. The two boundary segments are generally slightly smaller than the interior segments.
This strategy is suitable for algorithms which involve more computation at the boundaries of the data domain than in the interior.
Let ln denote the length of the interior segments and xl, xr the lengths of the leftmost and rightmost parts. The system tries to find a solution for ln, xl and xr such that

$$0.25 * ln \leq xl, xr \leq ln + 0.25 * ln$$

The value 0.25 is a predefined system parameter, but can be changed via the 'set' command (cf. *set eq* command in Appendix A). It restricts the solution in such a way that the boundary parts are selected so that they will not be much smaller than the inner parts. If the equational parameter is set to 0.0, this strategy guarantees that the boundary parts in the determined solution are smaller than the inner parts.
The system first attempts to fulfill the requirement that the boundary parts are smaller than the inner parts by searching for an integer solution to the following:

$$0.25 * ln \leq xl, xr \leq ln$$

If more than one solution exists, the solution with the smallest ln is selected. If a solution does not exist, the system tries to find a solution where the boundary parts are only slightly larger than the middle parts:

$$ln \leq xl, xr \leq ln + 0.25 * ln$$

If this also fails, the system returns with a corresponding error message.

Example: Let A(100) be an array.

```
part A(7 eq)
```

determines a partition of A according to the equal interior blocks strategy. The parts are:

```
A(1:13)   A(14:18)  A(29:43)   A(44:58)
A(59:73)  A(74:88)  A(89:100)
```

The segment lengths are 13, 15, 15, 15, 15, 15, 12.

3.2.1.4 General Block Strategy

The general block strategy permits the user to determine the number of parts as well as the length of the parts individually. They can either be specified explicitly by a positive integer number or implicitly by a linear expression over distribution variables.
Distribution variables are scalar variables which are undefined when the parallelization is started. They are not related to any variables declared in the program and need not themselves be declared. They are created at the time of their first use in a specification. They can be used to achieve alignment of variables as given in Vienna Fortran: for an example of this, see 3.2.2.1 below.

If a partition is specified using distribution variables, the system tries to find values for them by searching for a solution to the corresponding linear equation. If such a solution exists, the determined values are assigned to the corresponding variables and the lengths of the parts of the array are fixed accordingly. If a value has been determined for a distribution variable, this value can never be changed.

Example: Let A(100) be an array.

```
part A(2<7>-3<i>-3<j>-2<i-1>)
```

determines a partition of A according to the general block strategy. The variables i and j are distribution variables. The system determines the following solutions for i and j:

```
i=        17        14        11         8         5         2
j=         1         6        11        16        21        26
```

If the solution (17,1) is selected by the user, A is distributed as follows:

```
A(1:7)      A(8:14)    A(15:31)   A(32:48)   A(49:65)   A(66:66)
A(67:67)    A(68:68)   A(69:84)   A(85:100)
```

The segment lengths are 7, 7, 17, 17, 17, 1, 1, 1, 16, 16.

3.2.2 Example Distributions

The following show some uses of the interactive data distribution language.

3.2.2.1 Example: General Block Strategy

Let X(500), Y(500) and Z(511) be three arrays in a program.

```
part X(4)
part Y(4)
part Z(4)
```

distributes the arrays according to the default strategy.

```
processor 1:    X(1:125)       Y(1:125)      Z(1:127)
processor 2:   X(126:250)     Y(126:250)    Z(128:255)
processor 3:   X(251:375)     Y(251:375)    Z(256:383)
processor 4:   X(376:500)     Y(376:500)    Z(384:511)
```

If, however, the code consists of the loop:

```
      DO 123 I = 1, 500
           X(I) = Y(I) + Z(I+11)
 123  CONTINUE
```

it may be better to distribute the arrays according to this general block strategy:

```
pas p1(4)
part X(4<i>) with p1
part Y(4<i>) with p1
part Z(1<i+11>-3<i>) with p1
```

The distribution of the arrays will then be as follows:

```
p1(1):    X(1:125)       Y(1:125)      Z(1:136)
p1(2):   X(126:250)     Y(126:250)    Z(137:261)
p1(3):   X(251:375)     Y(251:375)    Z(262:386)
p1(4):   X(376:500)     Y(376:500)    Z(387:511)
```

Hence there is no communication required to execute the loop. Note that the distribution variable enables alignment of the arrays.

3.2.2.2 Example: Partial Replication

The following distribution was chosen for a program based on matrix multiplication. The code contained a nested DO-loop to perform the multiplication:

```
C = A * B
```

where A,B and C are identically-sized arrays representing three square matrices.
The selected distribution was chosen to eliminate communication during the multiplication at the cost of some additional storage space required to partially replicate arrays A and B:

```
pas  P(x,x)
part C(x,x) with P
part A(x,1) with (i=1,x (j=1,x A(i,1) -> P(i,j) ))
part B(1,x) with (i=1,x (j=1,x B(1,j) -> P(i,j) ))
```

First, a two-dimensional square processor array is declared. The result matrix C is distributed to this in roughly equal-sized square blocks using the default strategy. Array A is partitioned into roughly equal-sized segments of rows, since the second dimension is "divided" into one part[1]. These segments are then replicated across a row of processors. Similarly, matrix B is divided into segments of columns which are then replicated across the corresponding column of processors.

Assume that $x = 2$ and that each of the arrays have $100 * 100$ elements. Then the arrays will be distributed to the processors as follows:

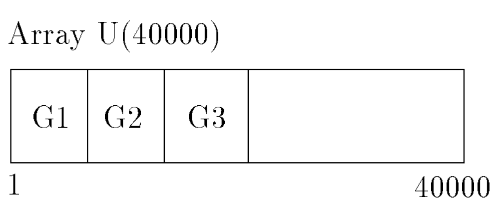
```
p(1,1):   A(1:50,1:100)     B(1:100,1:50)     C(1:50,1:50)
p(2,1):  A(51:100,1:100)    B(1:100,1:50)     C(51:100,1:50)
p(1,2):   A(1:50,1:100)    B(1:100,51:100)    C(1:50,51:100)
p(2,2):  A(51:100,1:100)   B(1:100,51:100)    C(51:100,51:100)
```

3.2.3 Workspace Concept

Many applications written in FORTRAN 77 use so called 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.

Example:

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 grids are mapped during execution of the program to the work array in the following way:



$G1$, $G2$ and $G3$ are accessed via dummy arrays which are bound at run time to single grids. Thus, the grids or, in the terminology of VFCS, the *virtual arrays* are represented in the program by these dummy arrays.

The following code excerpt demonstrates how the virtual arrays are accessed within the program:

```
DO 20 K=1,3
...
IDF=GSTART(K)
NPF=NPOINTS(K)
CALL RELAX (...,U(IDF),NPF,...)
...
 20 CONTINUE
```

[1]Note that this is how you may specify that an array dimension not be distributed in the distribution language.

```
SUBROUTINE RELAX (...,G,NP,...)
    REAL G(NP,NP)

...
END
```

The array element GSTART(K) contains the index of the element of U which is bound to the first element of the k-th grid. NPOINTS(K) contains the number of points in each dimension of the k-th grid. When the CALL statement is executed for the $k-th$ time, the dummy argument is thus associated with the $k-th$ grid. Inside subroutine RELAX, array G always refers to exactly one of the grids.

VFCS provides support for the workspace concept, by enabling the user to declare an array to be a workspace and distribute the individual parts of the workspace according to the requirements of the program. Workspaces and their virtual arrays (i.e. the arrays mapped in a workspace) are defined by means of the "ws" command (cf. A).

The "ws" command declares an array of the program to be a workspace; furthermore, the virtual arrays and their sequence in the workspace are defined. Data distribution is applied to the virtual arrays; different virtual arrays may be distributed in a different manner. Note that this distribution is static in the current implementation.

We return to the above example:

Let array U and arrays $G1$, $G2$ and $G3$ be used in a program as in the above multi-grid example. U is defined to be a work array which includes the virtual arrays $G1$, $G2$ and $G3$ by the following command:

```
ws U: G1(129,129), G2(65,65), G3(33,33)
```

In the data distribution language, the name of an array is qualified by the name of the unit in which it is declared and is not a dummy argument (this guarantees unique names in a program). Within the interactive facility, the arrays in the active unit may be distributed; a command is provided to activate a unit to enable a complete distribution of the program's arrays to be read from a single file. However, virtual arrays are not declared and hence do not belong to any unit. So that they may be distributed, they are considered to belong to "the system", and their names are qualified by the name **SYS** when the distribution is specified. Thus the work array in the previous example is fully identified by $MAIN.U$ and the virtual arrays by $SYS.G1$, $SYS.G2$ and $SYS.G3$.

The second step in the example consists of distributing the virtual arrays. For instance:

```
part SYS.G1(4,4)
```

partitions $G1$ into 16 two-dimensional segments.

3.2.3.1 Example: A Workspace Distribution

complete!!

3.2.4 Interactive Specification of FORALL Loops and Work Distribution

FORALL loops may be marked by selecting a DO loop in the program and the entry *Mark Parallel Loops* in the *Start* menu. The keyword *DO* is replaced in the program text by the keyword *FORALL*. Restrictions on loops which may be selected, and the manner in which they are processed is described elsewhere. The user is provided with information on the use of arrays and variables within this loop. If the system cannot prove that 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 FORALL loop.

These loops are initially provided with a work distribution by the system. Since this may not be the most suitable distribution, the interactive system enables you to modify the distribution via the *Modify Work Distribution* entry in the *Parallelize* menu. This may not be selected before the *Start 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, or in a block or cyclic fashion. The latter options are often favorable for short loops.

3.2.5 Batch Files for Data Distribution

Distribution files may be created and used to specify a data distribution. If these have the standard suffix *.spec*, they will also be recognized as such by a batch file when it parallelizes a program in batch mode. Files with the suffix *.spec* or *.part* may be created by the system to describe a data distribution: the user may also use these suffices, however, since they are not reserved for system use.

Distribution files may be created by the user

- interactively via the data distribution window, where there is an option to *Save* the distributions interactively entered to a file, or
- by manually writing the appropriate file in the interactive data distribution language.

Each program unit must be active when its local data is distributed. In the interactive system, this is accomplished by selecting a unit within the *Units* menu before entering the distribution commands. When a data distribution file is written, this is achieved by "loading" a unit before its local data is distributed. This is expressed by using the *load* command from the batch facility (see Appendix B). Examples of data distribution files are given in the demo codes.

3.3 General Constraints on Distributions

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.
First, 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.
It is also 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 can not find a way to remove the conflict, it will also report an error.
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 6.4.1.1.

Chapter 4

The Basic Parallelization Strategy

4.1 Overview

In this chapter we describe the basic steps of the parallelization strategy implemented in VFCS 1.0 by specifying a **source-to-source** translation from Vienna Fortran to a Fortran 77 enhanced with message-passing constructs (MPF). (We will not be concerned with the details of code generation for the different target architectures). We discuss the transformation steps only for those language features which are included in the current version of the system. In particular, the following assumes all arrays to be distributed **statically**, in the sense that once an array has been bound to a distribution (in a declaration), this association remains invariant throughout the entire program. Dummy arrays in **procedures** inherit the distribution of the actual argument.
The compilation strategy enforces the **owner computes paradigm**. That is, except for FORALL loops, we consider the **work distribution** to be derived automatically from the data distribution specified in the Vienna Fortran program.
In the remainder of this section we outline the overall program transformation strategy as a series of phases and discuss the methods for handling communication and storage allocation. The analysis of distributions at procedure boundaries is discussed separately. We then consider the strategy employed for adapting Vienna Fortran FORALL loops. This strategy has been implemented using the Parti routines, which are also briefly discussed.

4.2 The Parallelization Phases in VFCS

The translation from Vienna Fortran to MPF will be described as a sequence of **phases**, each of which specifies a translation between two source language levels (cf. Figure 4.1). We use the Jacobi demonstration code delivered with the system (program *jacobi*) to illustrate some of the transitions involved[1]. In particular, we will frequently refer to the Jacobi relaxation from that code to illustrate some of the details of the adaptation: we assume the form given in Figure 4.2.
Assume that a Vienna Fortran source program Q^0 is given. We transform Q^0 into an MPF program Q^4 in four conceptually distinct consecutive phases:

4.2.1 Phase 1 (Front End): Vienna Fortran Source Program $Q^0 \mapsto$ Vienna Fortran Normalized Program Q^1

The source program Q^0 is processed by the **Front End**, which performs the following three tasks: it transforms the source code into an *internal representation* suitable for further compiler manipulation; it performs the *initial*

[1] In the accompanying demo, $M = 4$ and $N = 1000$. The distribution is specified in the interactive language.

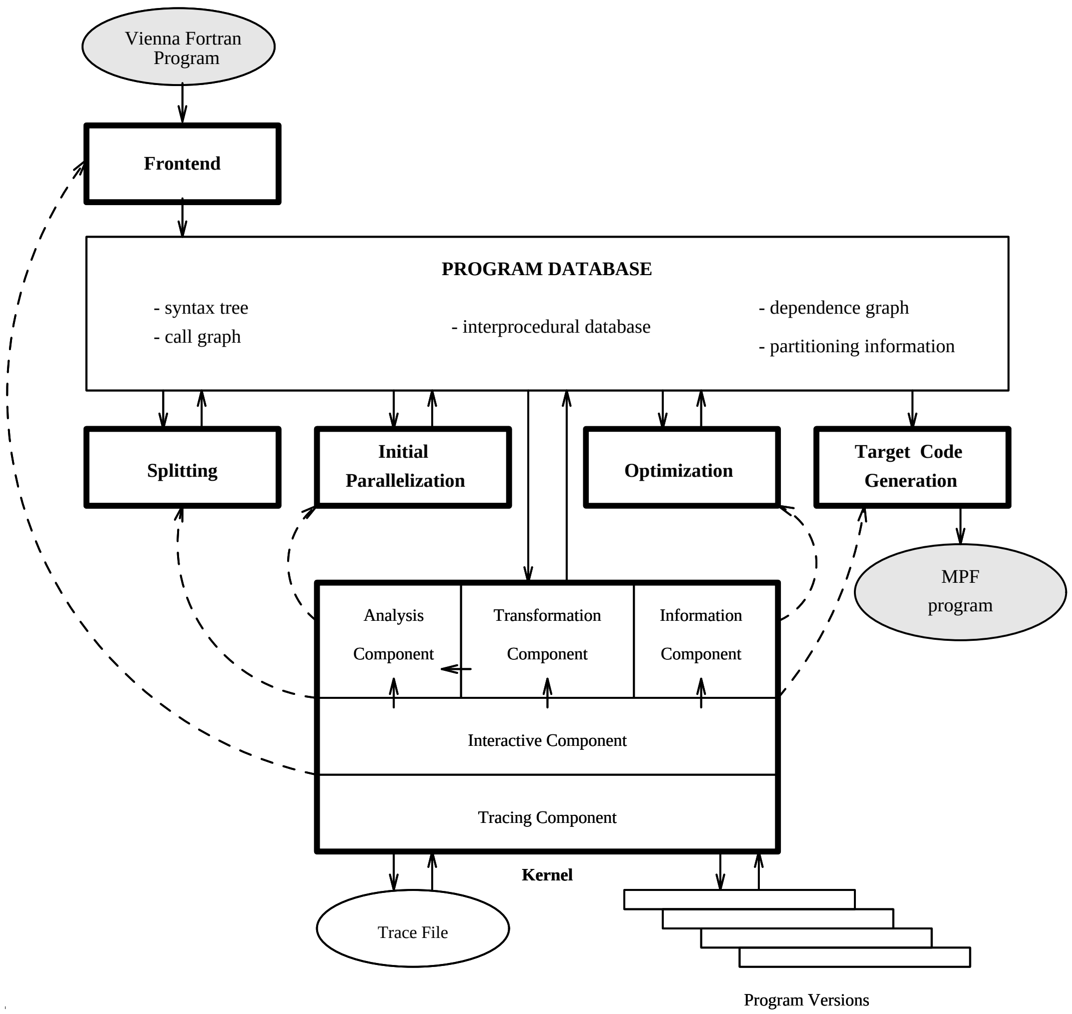


Figure 4.1: The Structure of VFCS

analysis of the program, and it *normalizes* the code. This analysis may have to be updated or recomputed during subsequent transformation phases.

Thus Phase 1 includes syntactic and semantic analysis, control flow analysis, data flow analysis, data dependence analysis and the construction of the call graph. The call graph is annotated with information needed in Phase 2. Further, some standard transformations such as constant propagation and dead code elimination are applied at this stage.

Normalizations have the purpose of simplifying subsequent compiler operations. Those implemented include transformations to eliminate certain program constructs (such as statement functions, ENTRY statements or alternate RETURNS), to position specific kinds of statement at a prescribed place in the code (EQUIVALENCE, FORMAT and DATA statements), and to transform other statements so that they always appear in a standard form (DO loops, IF and GOTO statements). Subscript expressions are standardized. The number of different kinds of statements affecting control flow may be considerably reduced by such transformations.

These transformations, which are also applied during vectorization and shared-memory parallelization, are further discussed in Chapter 6; a detailed treatment can be found in [40]. The resulting program, Q^1, is called

```
PARAMETER (M=2,N=16)
PROCESSORS R(M,M)
REAL UHELP(1:N,1:N), U(1:N,1:N), F(1:N,1:N)  DIST(BLOCK,BLOCK)
  . . . . .
   DO   J = 2, N-1
     DO   I = 2, N-1
       S: UHELP(I,J) = 0.25 * (F(I,J) + U(I-1,J) + U(I+1,J) + U(I,J-1) + U(I,J+1))
     ENDDO
   ENDDO
```

Figure 4.2: Jacobi relaxation: Excerpt from demo program with Vienna Fortran distribution.

the **normalized program**.

4.2.2 Phase 2 (Splitting): Vienna Fortran Normalized Program $Q^1 \mapsto$ (Host Program Q^H, Node Program Q^2)

Q^1 is split into a **host program**, Q^H, and a **node program**, Q^2. 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 program. Functions and subroutines which cause I/O also appear in the host program in a rudimentary form: both are converted to subroutine calls without arguments.

The node program, Q^2, which is still a sequential Vienna Fortran program, contains the actual computation; input/output is represented by communication with the host program. All I/O statements are removed from the node program: reads are converted to the original read, a SEND statement in the host and a RECEIVE in the node program; writes require the original write statement, a RECEIVE statement in the host and a SEND in the node program.

Program control flow is modeled in both codes: hence the node program 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 program. 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 programs resulting from this phase are loosely coupled; all synchronization necessary between them at run time will be achieved through message passing.

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 (Figure 4.4) 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 (Figure 4.5) contains all computation, but is still sequential. It receives data from the host, and in the excerpt given, executes the loop *iter* times, sending two values to the host in each iteration. We show only subroutine *NORM* here, which transfers output to the host.

```
PROCESSORS R(2,2)
REAL UHELP(1:N,1:N), U(1:N,1:N), F(1:N,1:N) DIST(BLOCK,BLOCK)
REAL RES(1:N,1: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/RNORMO
   WRITE(6,*) I,' CONVERGENCE:', RKONV
   RNORMO = RNORM
ENDDO
END
```

Figure 4.3: Jacobi demo: excerpt from main program before splitting

4.2.3 Phase 3 (Initial Parallelization)

This phase transforms the node program Q^2 into the **defining parallel program** Q^3 by processing the data distribution specified in the original Vienna Fortran program. It does so in two steps, which are referred to as **masking** and **communication insertion**. 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. For all non-local data accesses, communication insertion generates communication statements which copy non-local data items to *private variables* of the processor.
According to the *SPMD* execution model, the compiler does not generate separate node programs for each processor. Instead, each will execute the same program, receiving its parameters and initial data from the host program.
The performance of code to which only these simple transformations are applied is likely to be very poor. It is the task of all subsequent program manipulations to convert this node program into a form which will run well on the target machine. Both the construction of this version of the program and the subsequent improvements are discussed in more detail below.

4.2.4 Phase 4 (Optimization and Target Code generation): Defining Parallel Program $Q^3 \mapsto$ Optimized Target Program Q^4

In this phase, the defining parallel program is transformed into an optimized parallel MPF target program, Q^4. Communication and masking are improved: Communication statements are moved out of loops and combined to perform aggregate communication where possible; the *strip mining* of loops [40] across the processors can be achieved in many cases by propagating the information in masks to the loop bounds. A prerequisite for many optimizations is precise flow and data dependence information, as gathered in Phase 1, and **overlap analysis**, which detects certain simple regular communication patterns and re-organizes communication based upon this information. Overlap analysis also helps determine the minimum amount of storage which must be reserved for each non-replicated data array in the memory of a node processor.

```
      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
C*******************************************
      SUBROUTINE HOSTRESID()
      CALL HOSTNORM()
      END
C******************************************
      SUBROUTINE HOSTNORM()
      INTEGER ITER
      REAL RNORM
      RECEIVE ITER, RNORM FROM NODE
      WRITE(6,*) ITER, ' RESIDUUM:', RNORM
      END
```

Figure 4.4: Host Program after Splitting

```
C****************************************
      PROGRAM NODE
      PROCESSORS R(2,2)
        . . . . .
      RECEIVE ITER, OMEGA FROM HOST
      IF (ITER .EQ. 0) GOTO 2
      CALL INIT(F, U, N)
         . . . . .
      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
C***************************************
  SUBROUTINE RESID(ITER,RNORM,RES,F,U,N)
         . . . . .
      CALL NORM(ITER, RNORM, RES, N)
         . . . . .
C***************************************
      SUBROUTINE NORM(ITER, RNORM, RES, N)
         . . . . .
      RNORM = 0.0
      DO J=2,N-1
        DO I=2,N-1
          RNORM = RNORM + RES(I,J)*RES(I,J)
       ENDDO
      ENDDO
      RNORM = SQRT(RNORM)/FLOAT(N-1)
      SEND ITER, RNORM TO HOST
      END
```

Figure 4.5: Node Program after Splitting

4.3 Initial Parallelization

Initial parallelization is the third of the phases described above. Recall that a program has been analyzed and normalized before this phase begins, and that it has been split into a host and a node program. We study the steps required to convert the node program into a simple parallel program in 4.3.1 and 4.3.2 below. First, however, we define some notation we will need below. Note that the notation introduced here and in the following section does not correspond precisely to that used within VFCS 1.0. The VFCS-internal notation is described in Chapter 5.
Let S denote a statement, *ref* an array element, and p a processor. Then

- The occurrences of variable names in the program text are classified as **definitions** or **uses**, depending on whether they are assigned to or their value is read. We denote the corresponding sets for a statement S by **DEF(S)** and **USE(S)**.
- **MY_PROC** returns the unique identifier of the executing processor.
- **OWNED(ref[,p])** is a boolean function which yields **true** iff the array element denoted by *ref* is owned by p. The default for p is *MY_PROC*.
- **MASTER(ref)** returns a uniquely defined processor p, for which *OWNED(ref,p)* holds. It is called the **master** of *ref*. The choice of p is system-dependent.

4.3.1 Masking

The first step in the *initial parallelization* of a program enforces the *owner computes paradigm*: **Masking** replaces each statement S of the program by the associated **masked statement**

IF *mask(S)* **THEN** S **ENDIF**

where

- *mask(S)* is *OWNED*($A(\mathbf{x})$) if S is an assignment statement of the form $A(\mathbf{x}) = \ldots$, where A is non-replicated and $\mathbf{x}$ is an associated list of subscript expressions
- *mask(S)*$\equiv$**true** in all other cases. Clearly, in this case the **masked statement** associated with S can be immediately replaced by S.

Note that *mask(S)*$\equiv$**true** for all statements which are not assignments (in particular, control statements) and for all assignments to replicated variables.

Example 4.3.1:

The masked version of the assignment statement in the Jacobi relaxation (Figure 4.2) is: ∎

```
IF OWNED(UHELP(I,J))
 THEN UHELP(I,J)=0.25*(F(I,J)+U(I-1,J)+U(I+1,J)+U(I,J-1)+U(I,J+1))
ENDIF
```

4.3.2 Communication Insertion

The intermediate program version after masking may contain **non-local references**, i.e., a statement instance to be executed on one processor may refer to data owned by another one. Whenever this occurs, the required values must be communicated by message passing. This is the next step of parallelization.

Communication Primitive *EXCH0*

Processor communication in the *defining parallel program* Q^3 uses the **exchange primitive of level 0, EXCH0**, defined in Figure 4.6. *EXCH0* generates exactly those message passing statements that are required to satisfy the non-local data references in a processor's code. Each execution of a *SEND ref TO* p' in a processor p, which is caused by a call to *EXCH0(m,ref,temp)*, corresponds to exactly one execution of *RECEIVE temp FROM p* in p', caused by a call to the same occurrence of *EXCH0(m,ref,temp)*, in the same state.

/* **EXCH0(m,ref,temp)**
Assume that S is a statement, $m = mask(S)$, and *ref* is in *USE(S)* and of the form $A(\mathbf{x}) = \ldots$, where A is non-replicated. Furthermore, let *temp* denote a private variable of the executing processor. **MVAL(mask(S),p)** yields **true** iff the value of *mask(S)* in the current state is **true** for processor p.
The algorithm below specifies the effect of executing a call to *EXCH0(m,ref,temp)*. */

```
IF OWNED(ref)
  THEN temp := ref;
         IF  (MY_PROC = MASTER(ref))
         THEN FOR EVERY p SUCH THAT MVAL(m,p) ∧ (ref∉ λ(p)) SEND temp TO p
                ENDFOR
         ENDIF
  ELSE
    IF ¬ OWNED(ref) ∧ MVAL(m,MY_PROC)
      THEN    RECEIVE temp FROM MASTER(ref)
    ENDIF
ENDIF
```

Figure 4.6: The EXCH0 primitive.

Transformation 1 Communication Insertion

For every statement S with $m = mask(S)$, *and every* ref $\in$ USE(S) *of the form* $A(\mathbf{x}) = \ldots$, *where A is non-replicated, insert* **CALL EXCH0(m,ref,temp)** *before S. In S, replace* ref *by* temp. EXCH0(m,ref,temp) *is said to be* **associated** *with S. The order of different EXCH0 statements associated with a statement S is irrelevant; however, their private variables must be distinct.* □

For the following, we require additional notation:

Definition 4.3.1: Non-Local USE Sets

> *Let S denote a statement, p a processor, and* ref $\in$ USE(S). *Then* **NU(S,ref)** *denotes the set of all* **non-local uses** *caused by an access to* ref *when executing S in any processor, and* $\mu(p)$ *the set of all non-local accesses in processor p. Furthermore, we define* $\mu^A(p)$ *as the subset of* $\mu(p)$ *that contains only elements of A.* ■

Example 4.3.2:

> *We insert communication statements in the Jacobi relaxation loop (Figure 4.2). No exchange statement is generated for the reference to* $F(I,J)$. *Since* UHELP *and F are identically distributed and the subscript tuples associated with them are also identical in S,* NU(SI,F(I,J))= ϕ *for each instance of statement S. This can be detected by simple compiler analysis.*

The same does not hold for other uses in S. Consider, for example, the statement instances for which $I = 9$ and $2 \leq J \leq 8$, executed on processor $R(2,1)$. The union of the associated sets NU(SI,U(I-1,J)) *is* $\{U(8,2), U(8,3), \ldots, U(8,8)\}$. *Thus the computation of values on the "upper boundary" of* UHELP(9:16,1:8) *requires non-local access to the partial row $U(8, 2:8)$, owned by $R(1,1)$. The statement instances executed on $R(2,2)$ will similarly require non-local access to $U(8, 9:15)$ on $R(1,2)$.*

Now let us consider statement $C1$. TEMP1 is a private variable that receives a copy of the actual value of $U(I-1,J)$ – whether or not $U(I-1,J)$ is local for the executing processor. Execution of $C1$ causes the following communication:

- *In all iterations (I,J) with $I = 9$ and $2 \leq J \leq 8$, $R(1,1)$ sends the value of $U(I-1,J)$ to $R(2,1)$, and $R(2,1)$ receives the value of $U(I-1,J)$ from $R(1,1)$.*
- *In all iterations (I,J) with $I = 9$ and $9 \leq J \leq 15$, $R(1,2)$ sends the value of $U(I-1,J)$ to $R(2,2)$, and $R(2,2)$ receives the value of $U(I-1,J)$ from $R(1,2)$.*

■

```
PARAMETER (M=2,N=16)
PROCESSORS R(M,M)
REAL UHELP(1:N,1:N), U(1:N,1:N), F(1:N,1:N)  DIST(BLOCK,BLOCK)
PRIVATE REAL TEMP1, TEMP2, TEMP3, TEMP4

     . . . . .

 DO J = 2, N-1
    DO   I = 2, N-1

    C1: CALL EXCHO(OWNED(UHELP(I,J)),U(I-1,J),TEMP1)
    C2: CALL EXCHO(OWNED(UHELP(I,J)),U(I+1,J),TEMP2)
    C3: CALL EXCHO(OWNED(UHELP(I,J)),U(I,J-1),TEMP3)
    C4: CALL EXCHO(OWNED(UHELP(I,J)),U(I,J+1),TEMP4)
        IF OWNED(UHELP(I,J)) THEN
         S: UHELP(I,J)=0.25*(F(I,J)+TEMP1+TEMP2+TEMP3+TEMP4)
        ENDIF

     ENDDO
 ENDDO

     . . . . .
```

Figure 4.7: Jacobi relaxation code after initial parallelization.

4.4 Optimization and Target Code Generation

The *defining parallel program* specifies exactly the work distribution and communication required by the input program. However, it would be very inefficient to actually compile and execute such a program on a parallel computer since communication involves only single data items and in general each processor has to evaluate the mask of a statement for all instances of that statement.
Hence the code must be optimized to eliminate masks, by suitably modifying the (processor-specific) bounds of DO loops where possible. Communication must be optimized by moving communication statements out of loops and sending vectors of data where feasible. Redundant communication should be eliminated. The analysis described below can help in all of these tasks.

4.4.1 Overlap Analysis

Overlap analysis is performed in the compiler to determine which non-local elements of a non-replicated array 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.
For each non-replicated array A and processor $p \in P$, $\lambda^A(p)$ is the distribution segment of A in processor p; it is allocated in the local memory associated with p. In the defining parallel program, communication is inserted each time an element of $\mu^A(p)$ – the set of all non-local elements of A accessed in p – is referenced, and private variables are created to hold copies of the original non-local values (cf. 4.3.2). Overlap analysis is used to allocate memory space for these non-local values in locations adjacent to the local distribution segment.
More precisely, the **overlap area**, *OA(A,p)*, is the smallest rectilinear contiguous area around the distribution segment of a process p, containing all non-local variables accessed. The union of the distribution segment and the overlap area is called the **extension segment**, $\Lambda^A(p)$, associated with A and p. This description can be used 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 [11].
The overlap area for an array A is specified by its **overlap description**, $OD(A)$, which is determined by the maximum offsets for every dimension of the distribution segments, over all processors. If n is the rank of A, this takes the form

$$OD(A)=[dl_1 : du_1, \ldots, dl_n : du_n]$$

Here, dl_i and du_i denote the offsets with respect to the lower and upper bound of dimension i.
Finally, consider a statement S and *ref* $\in$ *USE(S)* with *ref*=$A(\ldots)$. The **overlap description** of S with respect to *ref*, *OD(S,ref)*, is defined as the contribution of S and *ref* to *OD(A)*.

Example 4.4.1:

> *Consider now the Jacobi code obtained above (Figure 4.7). The overlap descriptions associated with the different elements in* USE(S) *are computed as shown in Figure 4.8.*
>
> *Thus* $OD(F) = [0:0, 0:0]$*, and* $OD(U) = [1:1, 1:1]$.
>
> *The segment of* U *on* $R(1,2)$ *is* $U(1:8, 9:16)$. *Its extension segment is given by* $\Lambda^A(R(1,2)) = U(0:9, 8:17)$*, and the overlap area is* $U(0, 9:16) \cup U(9, 9:16) \cup U(1:8, 8) \cup U(1:8, 17)$. *It consists of an area of depth 1 around the distribution segment.*
>
> *Note that the overlap area may contain variables which do not correspond to elements of the original arrays. They will never be accessed, as they are not related to non-local uses.*

∎

We outline the computation of overlap descriptions and areas by considering the one-dimensional case.

- $OD(S, F(I, J)) = [0:0, 0:0]$
- $OD(S, U(I-1, J)) = [1:0, 0:0]$
- $OD(S, U(I+1, J)) = [0:1, 0:0]$
- $OD(S, U(I, J-1)) = [0:0, 1:0]$
- $OD(S, U(I, J+1)) = [0:0, 0:1]$

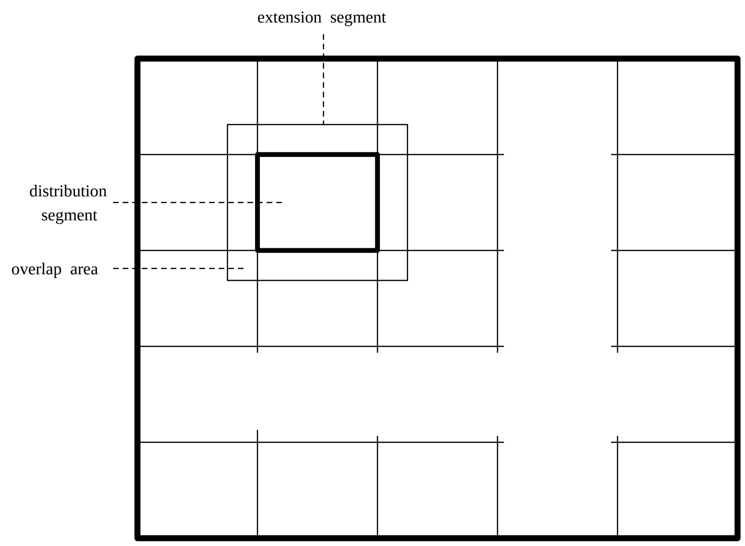


Figure 4.8: Overlap area for array U in Jacobi code ($M \geq 4$)

Consider an assignment statement $S : B(\mathbf{y}) = \ldots A(\mathbf{x}) \ldots$ in a loop with loop variable I and assume that A, B are non-replicated and that $\mathbf{x}$ and $\mathbf{y}$ are linear expressions in I.

1. Assume that $\delta^A = \delta^B$ and $\mathbf{x} = c_1 * I + d_1, \mathbf{y} = c_2 * I + d_2$, where $c_1 = c_2$, and d_1, and d_2 are constants. Let $d := d_1 - d_2$. If $d \geq 0$, then *OD(S,A(*$\mathbf{x}$*))*$=[0:d]$; otherwise *OD(S,A(*$\mathbf{x}$*))*$=[-d:0]$.

2. Assume now that $\delta^A = \delta^B$, $\mathbf{x}$ and $\mathbf{y}$ as above, but $c_1 \neq c_2$. Then, if a constant d is obtained from the symbolic subtraction $\mathbf{x} - \mathbf{y}$, we can proceed as above; otherwise, we apply the test below.

3. The general test is applicable to arbitrary distributions δ^A, δ^B. It constructs a set of inequalities that have to be satisfied by the loop variable for the instances of S associated with a process $p \in P$. This is based on $\mathbf{y}$ and the bounds of the distribution segments $\lambda^B(p)$. For example, if $\mathbf{y} = c_2 * I + d_2$ and $\lambda^B(p) = B(l_p : r_p)$ then $l_p \leq c * I + d \leq r_p$ must be satisfied for each p. The corresponding set of constraints for the loop variable can be determined by applying the Fourier-Motzkin method [21, 8] to these inequalities. From these constraints, $\delta^A(p)$, and $\mathbf{x}$, the overlap description *OD(S, A(*$\mathbf{x}$*))* can be computed.

4. 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. In the worst case, we obtain an extension segment with $\Lambda^A(p) = A$.

Hence 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 4.6.

4.4.2 The Communication Primitive *EXCH*

The overlap concept can be used to organize communication: the exchange statement *EXCH0* is replaced by *EXCH*, which refers to overlap descriptions *OD(S,ref)* rather than statement masks. *EXCH* is defined in Figure 4.9.

/* **EXCH(ref,od)**
Assume that S is a statement, $ref \in USE(S)$, where *ref* is of the form $A(\ldots)$ with A non-replicated, and $od{=}OD(S,ref)$. The algorithm below specifies the effect of executing a call to *EXCH(ref,od)*. */

```
IF  OWNED(ref) ∧ ( MY_ PROC = MASTER(ref))
   THEN FOR EVERY p SUCH THAT  ref ∈  OA(A,p) SEND  ref TO p ENDFOR
   ELSE
     IF  ref ∈  OA(A,MY_ PROC)
        THEN    RECEIVE  ref FROM  MASTER(ref)
     ENDIF
ENDIF
```

Figure 4.9: The EXCH primitive.

4.4.3 Optimization of Communication I

Transformation 2 Communication Insertion Based on Overlap

For every statement S *and every* ref $\in$ USE(S) *with* ref=A(...), *A non-replicated, and* od=OD(S,ref), *insert* **CALL EXCH(ref,od)** *before* S. *The order of different EXCH statements associated with the same statement* S *is irrelevant.* □

We replace Transformation 1 by Transformation 2. The new version of *Jacobi* is given in Figure 4.10. The local declarations are omitted (see Section 4.4.5).

```
          . . . . .

DO J = 2, N-1
   DO   I = 2, N-1
   C1A: CALL EXCH(U(I-1,J),[1:0,0:0])
   C2A: CALL EXCH(U(I+1,J),[0:1,0:0])
   C3A: CALL EXCH(U(I,J-1),[0:0,1:0])
   C4A: CALL EXCH(U(I,J+1),[0:0,0:1])
       IF OWNED(UHELP(I,J)) THEN
        S: UHELP(I,J)=0.25*(F(I,J)+U(I-1,J)+U(I+1,J)+U(I,J-1)+U(I,J+1))
       ENDIF
    ENDDO
ENDDO

          . . . . .
```

Figure 4.10: Program Jacobi with EXCH-based communication

4.4.4 Optimization of Communication II

The modified exchange statement forms the basis for a more advanced optimization of communication: we generalize *EXCH* to an **aggregate communication primitive** that moves blocks of data rather than single objects, and then discuss how to move communication out of loops and vectorize communication statements.

EXCH can be readily extended to move any rectilinear contiguous section $\hat{A}$ of a segment $\lambda^A(p)$ by replacing *ref* in Figure 4.9 with $\hat{A}$, and taking care that more than one processor may be the source of a *RECEIVE*. The communication for a data item can be moved out of a loop if no true dependence is violated. Loop distribution and vectorization [40] can then be applied to generate aggregate communication. Further optimization of communication includes **fusion** and **elimination of redundant communication**, as described in the literature [11, 23].

4.4.5 Optimization of Masking

After initial masking (Section 4.3.1), all processors execute the same masked statement sequence. For each masked statement, each processor first evaluates the mask and executes the corresponding (unmasked) statement instance if the mask yields **true**. 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.
2. **Mask Simplification**: If it can be shown that a mask is always **true** for each instance of each process, it can be eliminated.

```
      PARAMETER (N=16)

      REAL UHELP($L1(p)-1:$U1(p)+1,$L2(p)-1:$U2(p)+1)
      REAL U($L1(p)-1:$U1(p)+1,$L2(p)-1:$U2(p)+1)
      REAL F($L1(p):$U1(p),$L2(p):$U2(p))
            . . . . .

  C1T: CALL EXCH(U(1:N-2,2:N-1),[1:0,0:0])
  C2T: CALL EXCH(U(3:N,2:N-1),[0:1,0:0])
  C3T: CALL EXCH(U(2:N-1,1:N-2),[0:0,1:0])
  C4T: CALL EXCH(U(2:N-1,3:N),[0:0,0:1])
   DO J = MAX(2,$L1(p)),MIN(N-1,$U1(p))
      DO   I = MAX(2,$L2(p)),MIN(N-1,$U2(p))
        S: UHELP(I,J)=0.25*(F(I,J)+U(I-1,J)+U(I+1,J)+U(I,J-1)+U(I,J+1))
      ENDDO
   ENDDO

            . . . . .
```

Figure 4.11: Jacobi – Final Version: Code for Processor p.

A more general discussion of mask optimization and the associated work distribution can be found in [14]. We again illustrate these optimizations as well as the target code generation with the example program.

Figure 4.11 gives the final version of the Jacobi iteration. The program is parameterized in terms of the executing processor, p. It is assumed that $\lambda^U(p) = U(\$L1(p):\$U1(p), \$L2(p):\$U2(p))$. Hence, for example, $\$L2(R(1,2)) = 9$ and $\$U2(R(1,2)) = 16$. The local declarations reserve space for the extension segments of $UHELP$ and U. To simplify matters, we assume here that the target language can handle expressions in array bound declarations, so that global addressing is possible. (In VFCS 1.0, this is accomplished by an appropriate initialization of the node processes when execution begins).

The execution of the first exchange statement, *CALL EXCH(U(1:N-2,2:N-1),[1:0,0:0])*, in a processor p has the following effect: first, all elements of *U(1:N-2,2:N-1)* that are owned by p and belong to the corresponding overlap area of another processor p', are sent to p'. Secondly, all elements of *U(1:N-2,2:N-1)* that are in *OA(U,p)* are received from the respective owner. For example, $R(2,1)$ sends *U(9:14,8)* to $R(2,2)$, and $R(1,2)$ receives *U(1:8,8)* from $R(1,1)$. These communication operations can be executed in parallel.

For each p the set of associated instances of S is given by $\{S(I,J) \mid MAX(\$L1(p),2) \leq I \leq MIN(\$U1(p), N-1)$ and $MAX(\$L2(p),2) \leq J \leq MIN(\$U2(p))\}$. For these instances, the mask can be eliminated; all other iterations can be eliminated for p.

4.5 Procedures

In this section, we discuss the handling of procedures and their formal scalar and array arguments. We apply the notation to describe local variables to formal procedure arguments without further introduction. The *SPMD paradigm* is applied to procedures in an obvious way: a procedure call in the node program is executed by all processors allocated to the program; masking and communication is performed as described above. We assume here that the distributions of dummy arguments are **inherited** from the corresponding argument distributions. The quality of the code generated for procedures then depends critically on how much information is available at compile time about the distributions of formal arguments in different incarnations of the procedure, and on how efficiently run-time information can be organized. We use the term *block distribution* in this Section to denote any distribution in which each processor owns one contiguous segment of elements of an array. Thus a distribution constructed using any combination of BLOCK, S_BLOCK, B_BLOCK, the additional interactive strategies described in 3.2.1, and replication is a block distribution. In Example 4.5.1, we show even if all arrays in a program have block distributions, this does not necessarily imply that dummy arguments have a block distribution.

Example 4.5.1:

Consider the program in Figure 4.12. The processor array, R, and U are declared in the same way as in the Jacobi *program, and U has the same distribution (see Figure 4.2).*

The first subroutine call maps the two-dimensional array U to the one-dimensional array X. The sequence association defined in Fortran means that $U(i,j)$ is associated with $X(i+(j-1)*16)$ for all i,j with $1 \leq i,j \leq 16$. Now let p denote the processor $R(1,1)$. Since $\lambda^U(p) = U(1:8,1:8)$, we obtain $\lambda^X(p) = X(1:8) \cup X(17:24) \cup \ldots \cup X(113:120)$ – thus, $\lambda^X(p)$ is not a segment, but rather a union of eight disjoint segments. Similar observations can be made for the other processors. As a consequence, X does not have a block distribution.

In contrast to this, the Ith call to the second subroutine maps the I-th column of U, $U(1:16,I)$, to Y; that is, for every $k, 1 \leq k \leq 16$, $U(k,I)$ is associated with $Y(k)$. The distribution of Y is thus as follows: if $1 \leq I \leq 8$, then

- $\lambda^Y(R(1,1)) = Y(1:8)$, $\lambda^Y(R(2,1)) = Y(9:16)$, $\lambda^Y(R(1,2)) = \lambda^Y(R(2,2)) = \phi$

If $9 \leq I \leq 16$, then

- $\lambda^Y(R(1,1)) = \lambda^Y(R(2,1)) = \phi$, $\lambda^Y(R(1,2)) = Y(1:8)$, $\lambda^Y(R(2,2)) = Y(9:16)$

In both cases Y has a block distribution; note however, that some segments may be empty.

■

Dummy arrays that do not have a block distribution cannot be handled as efficiently as those that do, since masks can no longer be efficiently implemented, and the related optimizations cannot be performed in general. Note, too, that different incarnations of the same procedure may associate the same dummy argument with arrays that have different distribution characteristics. The code implementing the accesses to a formal array argument has to take into account all possible cases for the corresponding argument.

Gerndt [11] has formulated a sufficient condition (using the concept of a *subarray*) to guarantee that the dummy argument associated with a block-distributed actual argument is also distributed by block. This is used below to optimize the code generation for procedures.

```
*** DUMMY argumentS --- NODE PROGRAM ***

  PROCESSORS R(2,2)
  REAL  U(1:16,1:16)  DIST(BLOCK,BLOCK)

        . . . . .

  S1: CALL SUB1(U)
        . . . . .
      DO   I = 1,16
S2: CALL SUB2(U(1,I))
      ENDDO
        . . . . .
```

```
SUBROUTINE SUB1(X)
REAL X(1:256)
   DO   J = 1,256
   X(J)=X(J)*J
   ENDDO
END

SUBROUTINE SUB2(Y)
REAL Y(1:16)
   DO  K = 1,16
   Y(K)=Y(K)+1
   ENDDO
END
```

Figure 4.12: Distribution of dummy arguments – node program.

We describe the main features of the interprocedural optimization strategy for handling of distributed arrays in procedures as implemented in VFCS. More details are given in [12, 11]. We assume that a **call graph**, G, for the program has been constructed. G contains a node for each procedure of the program, and an edge from node n to node n' if the body of n contains a call to n'. Since we assume all procedures to be non-recursive, G can always be made acyclic (possibly by applying node splitting transformations). The transformations described below can be performed during a single pass over G in *invocation order*: thus, whenever a procedure is processed, all procedures that may execute a call to it have already been handled.

The following steps are taken to process a procedure T in the pass over G:

1. **Attribute Specification**
   For each dummy argument, f, of T, and each call site associated with T, the argument, a, with which f is associated is examined. If a is a non-formal object, then it is uniquely associated with the attribute "r" or "b", depending on whether a is **r**eplicated or (general) **b**lock distributed. If a is a dummy argument, then it has one or more of the attributes "r", "b", and "n", where "n" stands for **n**on-block distribution, and the presence of an attribute indicates the kind of distribution that may be associated with a. We now describe how the attributes of f are determined:[2]

   - If a has the attribute "r", then f is marked with "r"
   - If a has the attribute "b" and the association of f with a is such that block distributions are maintained, then f is marked with "b", else f is marked with "n"
   - If a has the attribute "n", then f is marked with "n"

2. **Cloning**
   If f is marked by more than one attribute, then a separate copy of T is created for each attribute associated with f, and the call graph as well as the call statements to T are correspondingly modified. This process is called **cloning** of T. In each clone, the attributes for each formal argument are unique.

3. **Expansion of Scalars**
   If f is scalar and is marked with either "b" or "n", then it is expanded into an array.

4. **Computation of Distribution Vectors**
   Assume that T has dummy arguments $f_1, \ldots, f_m$. Then for each incarnation of T, a **distribution vector** is computed. For each f_k, $1 \leq k \leq m$, if f_k is replicated or has a block distribution, the distribution vector specifies the distribution to which f_k is bound in the given incarnation, If it has a non-block distribution, the distribution of the corresponding actual array is determined.

[2]Note that more than one case may apply.

4.6 FORALL Loops and the Parti Routines

Codes where arrays are accessed indirectly (via index arrays) are common in many application areas. Since VFCS relies on compile-time analysis to allocate memory storage, these will in general lead to an inefficient use of memory. For example, given a loop such as

```
      DO 333  I = 1,N
          X(I) = X(I)+C(K(I))*Y(K(I))
 333  CONTINUE
```

the system cannot determine which elements of C and Y will be read by the processor which owns an element of X. This means that it will reserve storage space for the entire arrays C and Y on each processor. (If we were to replace the occurrence of X(I) on the left-hand side of the assignment by Y(K(I)), then the system could also no longer detect which processor should execute an iteration and this would also have to be tested for at run time.)

Vienna Fortran provides the FORALL loop, with which the user can support the system in its handling of such codes. We briefly outline the manner in which VFCS 1.0 handles FORALL loops below, and then discuss the Parti routines which are used by VFCS to realize this transformation.

FORALL loops are also often called *parallel loops*. Note that this terminology does not mean that the user can write machine-specific parallel constructs such as SENDs or a specific reduction operation for a target machine within them: only the constructs provided within Vienna Fortran may be used.

4.6.1 The Parallelization of FORALL Loops

Vienna Fortran **FORALL loops**[3] give the user the possibility to specify that the iterations of a loop can be executed independently in situations where compile time analysis cannot detect this. Iterations may be mapped to the processors in a number of ways, so that assignments may be made to non-local variables. Thus the code compiled for such loops differs significantly from code generated for other parts of the program.

VFCS integrates the parallelization of FORALL loops and conventional DO loops within a single program, so that if an array is accessed both regularly and by an index array, a FORALL 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 FORALL loops and to do so it must make certain assumptions about the amount of memory required for local storage of such data.

The general strategy employed to handle FORALL loops in VFCS is described in 6.6 (see also [42]). VFCS generates two sections of code, known as the **inspector** and the **executor**, for each FORALL loop. The inspector analyzes the communication patterns of the loop, while the executor performs the actual communication and executes the loop iterations. Hence in this context, the term **inspector-executor paradigm** is frequently used. For a general discussion of this paradigm, see [17].

In the following, we denote by *SENDS*(p,p') the set of elements that must be sent from processor p to processor p' as a consequence of non-local uses in p'. We let *RECEIVES*(p,p') be the set of elements that must be received by p from p' as a consequence of non-local uses in p. Clearly, *SENDS*(p,p')=*RECEIVES*(p',p) for each pair of processors. The set of iterations to be performed on processor p will be called *ITS*(p). We use the term *local iteration* for each iteration which does not use any non-local values; the set of these for a processor p is *LITS*(p). The remaining iterations are termed *non-local iterations*.

4.6.2 The Inspector

The **inspector** implements the first phase in the execution of a FORALL loop by analyzing the loop and computing the communication and iteration sets. Figure 4.13 specifies the code of the inspector executed in a

[3]Note that this is not the FORALL loop proposed during the development of Fortran 90.

processor p. We assume that the sets *ITS(p)* have been determined.

The inspector consists of two steps. The *first step* performs a dynamic loop analysis: a simplified version of the loop is executed, in which only those components are analyzed that are relevant for the generation of communication. For each iteration in *ITS(p)*, it is determined whether or not it is a local iteration in the above sense. For each non-local iteration, the non-local uses and their associated processors are determined: this yields the sets *RECEIVES(p,p')* for all $p' \in P$. The *second step* uses the relationship *SENDS(p,p')=RECEIVES(p',p)* to compute, as a result of a global communication phase, the sets of values which must be transmitted prior to execution of the loop.

Although this strategy is superior to individual communications of values, the inspector phase nevertheless represents a significant run time overhead.

INSPECTOR

Step 1: Loop Analysis
Perform a dynamic analysis of the loop to determine the local
and non-local iteration sets,and, for all p', RECEIVES(p,p').

Step 2: Computation of SENDS sets
In a global communication phase, compute all sets SENDS(p,p')
from the receive sets determined in Step 1.

EXECUTOR

Step 1: Send
For each $p' \in P$ such that SENDS(p,p')$\neq \phi$:
SEND SENDS(p,p') TO p'
Step 2: Execute Local Iterations
Execute all iterations with local data only.
Step 3: Receive
For each $p' \in P$ such that RECEIVES(p,p')$\neq \phi$:
RECEIVE temps FROM p'
Step 4: Execute Non-local Iterations
Execute all remaining iterations.

Figure 4.13: Inspector/Executor code for processor p.

4.6.3 The Executor

The **executor** is the second phase in the execution of the FORALL statement; it performs the actual computation. When the executor commences, all communication and iteration sets have been computed in the inspector. The executor consists of four steps (see Figure 4.13). The first of these sends all local data that are required by other processors. Then all local iterations are executed. In parallel (if the target machine permits this), non-local data required in one processor is received from the others and stored in local memory. Finally, the executor performs all iterations with at least one non-local use, accessing the corresponding local copies.
We have not shown the final stage of a FORALL loop: if values have been assigned to non-local data, then there must be a concluding communication phase which transmits these to all processors which own them.

4.6.4 The PARTI Routines

The VFCS implementation of the strategy outlined above for parallelizing FORALL loops is supported by the Parti routines: calls to these are inserted into the parallel program.

The **Parti** routines ([37, 34]) are a library of primitives constructed to support the handling of codes with indirect accesses and irregularly distributed arrays, whose distributions are specified via **mapping arrays**, where the value of a mapping array element is the number of the processor storing the array element with the same index. The mapping array is very costly in terms of storage space, since it is as large as the original array, however it permits the specification of an arbitrary distribution.

There are Parti routines to locate an element of an arbitrarily distributed array, to receive and transmit irregularly distributed non-local data, and perform reduction operations on them. They can be used to transform FORALL loops according to the strategy described in the preceding section.

In order to deal with array elements which are arbitrarily assigned to processors, a routine is called to construct a translation table from the mapping array, recording the owner of each datum and its local index. It is generally itself distributed in a regular fashion across the executing processors. In Step 1 of the inspector, communication has to be performed to find out which processor a data item is stored on, and get the local index. In view of the expense of these operations, both the Parti routines and the manner in which they are used by VFCS 1.0 have been optimized. Tables are constructed for classes of arrays where possible; local copies of non-local data are stored using hash tables in a so-called hashed cache. This helps eliminate duplicate fetches. Methods to keep track of the location of non-local data and reuse values already received have been developed on the basis of incremental communication schedules, which fetch only those data items which are not already available locally – this often proves useful for sequences of loops where reuse of data items is not uncommon.

The Parti routines have been designed so that they can be inserted into code by a compiler or used directly by programmers.

Chapter 5

Using VFCS 1.0

5.1 Introduction

In this chapter you will find the basic information you need to work with the system. The first section below gives an overview of how the system is used; the following section then describes the main components of the user interface and explains how to perform the most important operations. If you are new to VFCS, then you should read these sections first.
The rest of the chapter describes the VFCS menus in some detail. The main menu (cf. Fig. 5.1) contains a number of entries, each corresponding to a pull-down submenu (second-level menu). Each entry in the main menu is described in a separate subsection. Details of the operations underlying the menu items are deferred to subsequent chapters.
Before you start to work, it is important that you read the section at the end of this chapter on how to prepare your program for parallelization. Please refer also to the current list of bugs in the corresponding chapter of this document.

5.1.1 Modes of Operation

The Vienna Fortran Compilation System (VFCS) has two basic modes of operation, the batch (automatic) mode and the interactive (user-controlled) mode. This chapter is concerned primarily with the latter, which enables the user to actively participate in the process of parallelization and provides the tools required to identify and modify those parts of a program which are likely to cause poor performance or incur memory wastage in their original form. The batch mode is described in Chapter 7.

5.1.2 Getting Started

The Vienna Fortran Compilation System (VFCS) is invoked by typing

```
vfcs [batch_file]
```

at the command line.

If the batch_file is omitted, the interactive VFCS window is displayed. The VFCS main menu is presented in the second line of this window. Otherwise, the commands in the batch_file are executed without user interaction.

Note that the path for vfcs must be set in your local shell script before you first use the system. This is described in the Appendix E. Ask your system administrator for local details.

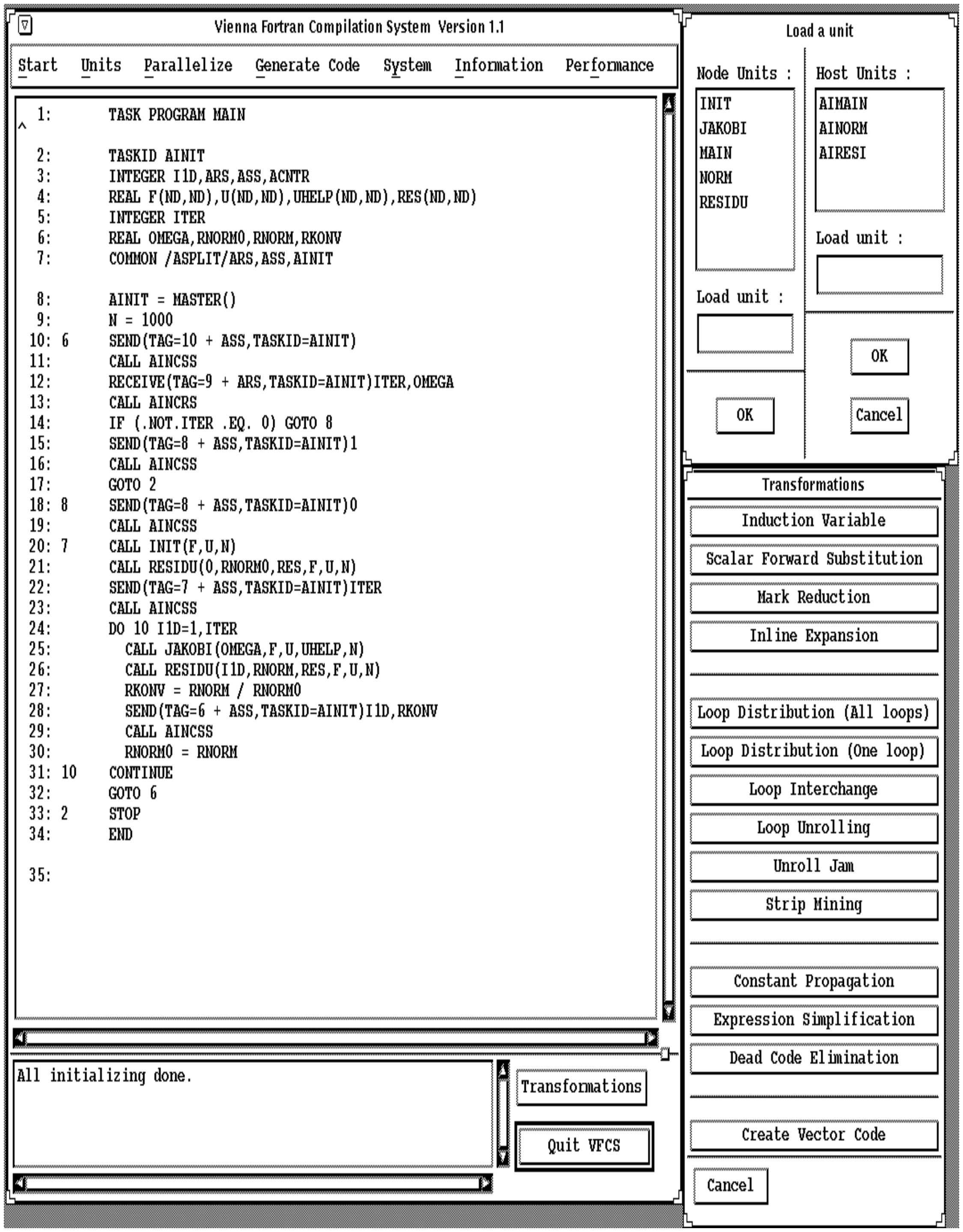


Figure 5.1: The main menu, unit selection box and transformation menu

5.1.3 Leaving The System

You may quit VFCS at any time by clicking on the button at the bottom right of the VFCS window.
Warning: The system does not automatically save any files when you quit. The appropriate menu entries (in the *Units* and *Parallelize* menus) must be selected to save your files.

5.1.4 Sequential Versus Parallel Processing

Although VFCS was primarily conceived as a system for converting FORTRAN 77 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. To permit this, there are separate options within the *Start* menu (described in 5.3.2 below) for selecting source code for parallel and sequential processing. Although the method by which the user specifies the appropriate files is the same in each case, the initial processing of these files differs.

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 from the *Transformations* menu.

5.1.5 The 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 transformations implemented within the system are the subject of a subsequent chapter.
The first step in parallelizing an application consists of selecting the files which make up the complete program. Note that VFCS 1.0 will only operate correctly on **entire programs**, although initial processing may be done on individual files and files may be replaced by updated versions on an individual basis.
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.
To select your source files, you must enter the *Start* menu and then select the menu item *Get Sources for Parallelization*.
The file selection box then appears on the screen so that you may locate and select the program files. The details of how to do this are to be found in subsection 5.2.2.3 below.
Once you have selected your source files, you may leave the file selection box by pressing the *OK* button. If the source program is incomplete, the system will inform you of this. You may then return to the file selection box to select the remaining files.
The system then automatically initiates the program analysis by reading each of the source files in turn and constructing an internal representation of the program; it also modifies certain Fortran constructs to make subsequent transformations easier to apply and more efficient. In addition it will create two separate programs to replace your original one: the host program will perform input and output, and the node program will perform the computation in the original code (cf. 4.2.2). To get a list of the units in your program, you must invoke the first item in the *Units* menu, which is described in subsection 5.2.2.4 below.

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 9. If you do not remove these from your source, the system will report a program error if it detects any during the initial processing.

If your source program is written in Vienna Fortran, and you do not wish to modify the data distribution or change any DO loops to FORALL loops, you may proceed directly to the *Parallelize* menu, where you should select the first menu item, *Start Parallelization*.

If your source program does not include a data distribution or have parallel loops marked, or if you want to check and possibly modify either of these, you should select the corresponding menu items in the *Start* menu first, in any order and as often as you wish, before proceeding to the *Parallelize* menu. These menus are described in Section 5.3 below; you may wish to select different program units in order to distribute their arrays or indicate the parallel loops. The way do this is also described in 5.2.2.4. Once this is completed, invoke the *Start Parallelization* menu item as above.
Note: A loop does not have to be marked as parallel in order to be executed in parallel. This should only be explicitly indicated in cases where the system will not be able to detect that different iterations of the loop are independent – in particular, when arrays are accessed indirectly (see 6.6).

After the *Start Parallelization* menu has been invoked, the program has been parallelized: a VFCS-specific notation is used to mark communication (5.1.6) in this *defining parallel program* and to give some indication of where statements will be executed. This version, when output by the back end, would run on the target machine, but it would probably be very inefficient.
To produce better code, select the next two items in the menu, *Optimize Communication* and *Optimize Masking*, in turn. Then select the entry *Mask Propagation* for each unit of the node program in turn. (These are not separate menu items because they require input from the user, but because the user is permitted to see the state of the program after each of these operations, and if desired, perform one or more transformations on the code). 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 sometimes 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.

You may now proceed directly to the *Generate Code* entry in the main menu. Enter this and select the first item to indicate your target machine. The back end for the machine will then run automatically to adapt your program to the specific message-passing constructs provided in the appropriate message-passing language for it. Save the generated code to disk by pressing the last item on the menu, *Save Parallel Program*.

Warning: The system does not automatically save the parallel program for you.

To parallelize a complete Vienna Fortran program without applying any modifications, you should:

- Select your source files for parallelization (cf. Fig. 5.2)
- Select the menu item *Start Parallelization*
- Select *Optimize Communication*
- Select *Optimize Masking*
- Select *Mask Propagation* for each unit of the node program containing communication (this may also be useful before selection of *Optimize Communication*, and is optional)
- Select your target machine
- Save your parallel program

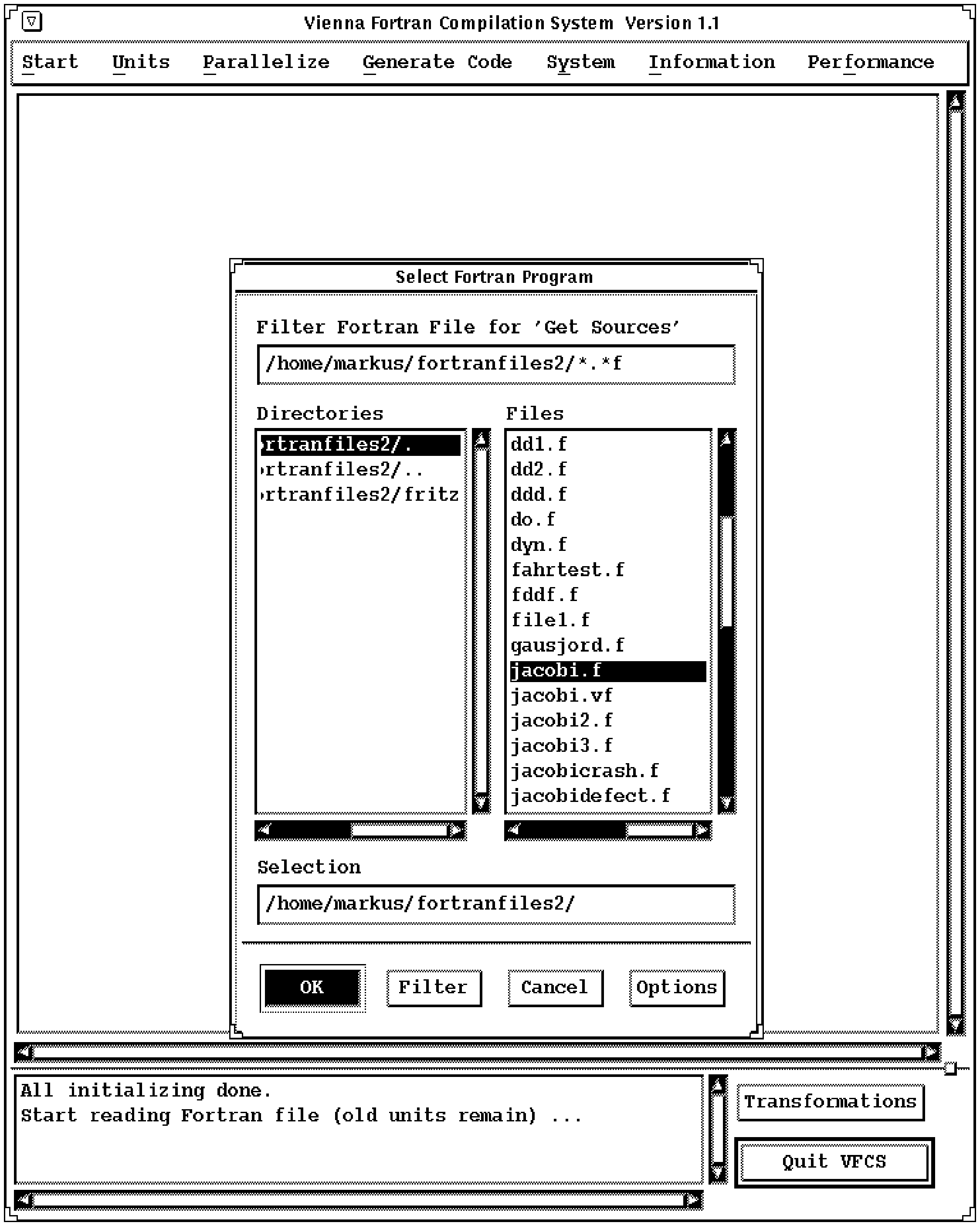


Figure 5.2: Selecting Fortran programs

5.1.6 VFCS Internal Language Elements

The effects of the parallelizing transformations and user-selectable transformations on the Fortran source code can be observed in the VFCS program window (see Fig. 5.3). 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.
Where a term is enclosed by "<" and ">", the term is to be substituted by a legal value. Those terms not enclosed are keywords which appear as written.

The following describe ownership of data, in particular to indicate where an iteration of a loop will be executed or which processor is responsible for sending data:

<mask>-> :

When prefixed to a program statement, the statement is only executed when the condition defined by <*mask*> is true. This restricts the number of processors which execute the statement.

Mask may be one of the following:

OWNED (<array element>) :
The executing processor is the one that owns the specified array element (4.3.1).

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 (4.3).

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

This notation is used to describe the overlap area computed and to indicate the exchange of data between node processors via explicit communication:

{[dl1/du1],..[dln/dun]} :
This describes the overlap area as defined in Section 4.4.1. It 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.

EXSR <array_reference> <ovp_area_descr> :
The EXchange Send Receive construct is used to represent the generalized EXCH communication primitive (see 4.4.2). It indicates the exchanging of values of <array_reference> between processors depending on the overlap description given in <ovp_area_descr>.

The parallelization of entire DO loops is indicated by the following notation:

mask-> <do-stmt>) :
If the optimization of masking is successful, the OWNED masks for the statements within the loop body are transformed into local loop bounds for the DO loop headed by the <do-stmt>. This is indicated by the above notation.

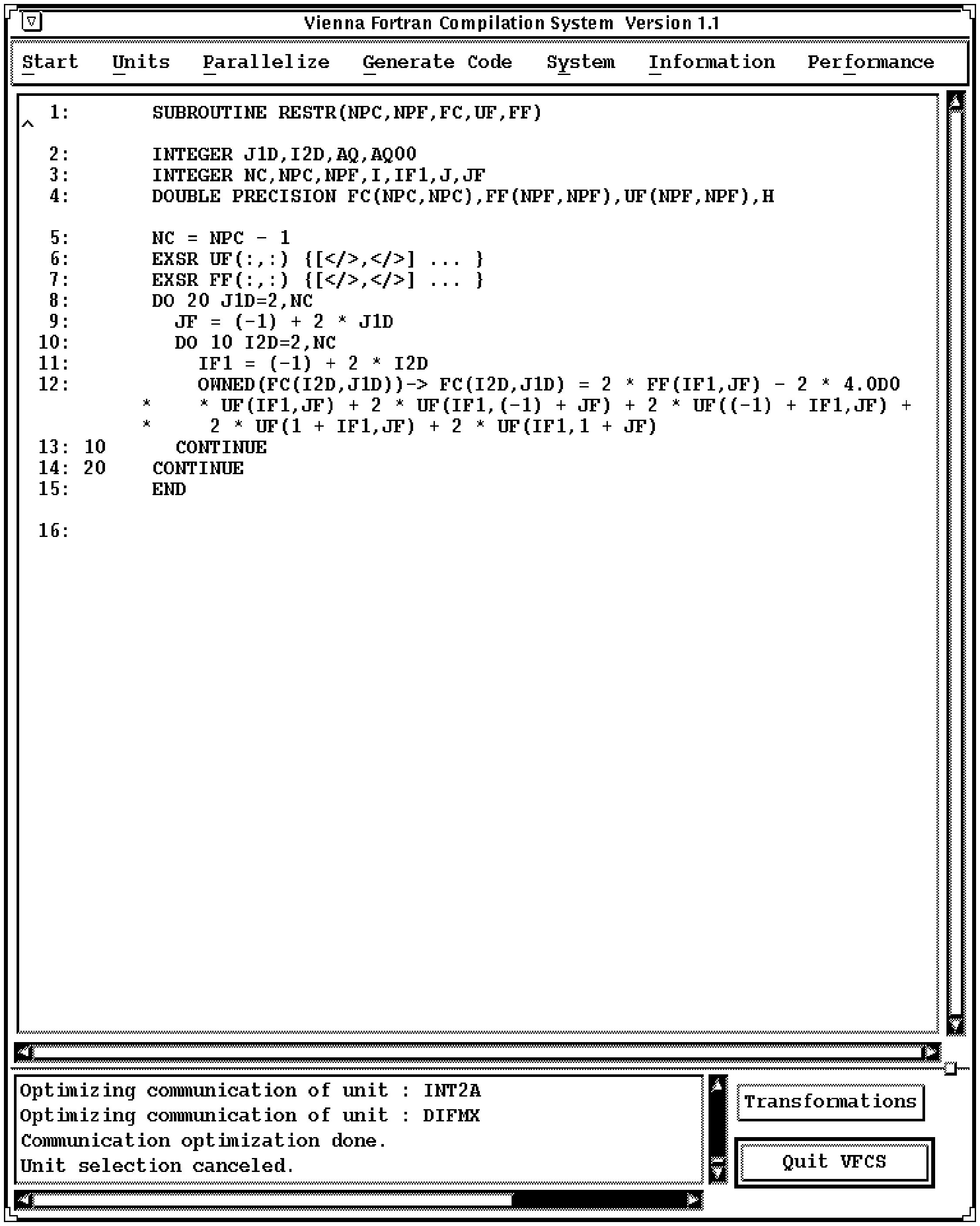


Figure 5.3: Fortran program with EXSR and OWNED statements

Interactively specified FORALL loops are marked with the following notation:

FORALL :
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.

5.1.7 VFCS Output Files

VFCS creates several files in your working directory. Some of them are protocols which are only of interest in case of error. The most important files are the following:

- Your host program is contained in the files *host.f* and *hostsys.f*
- Your node program is contained in the files *node.f* and *nodesys.f*
- Setup data for your program is contained in the files *init.**. The size and contents of these files depends on the structure of your program. They are read by the host program during execution.

These form the parallel program and must be copied to the target machine if they are not created there.

Other files include:

- **.part* files contain information on the data distribution. You may create these yourselves using the language constructs provided. **.spec* files also contain information on the data distribution, as created by the system from interactively specified distributions. Batch files expect distributions in files with these names.
- **.bat* files are batch files used to run VFCS in batch mode. **.pro* files are batch files created by the protocol facility. This topic is covered in Chapter 7.
- The *Units* directory is created to hold program information and, in particular, store all data to file needed to recreate program versions that you define during a parallelization session. Do not modify any files in this directory. If you wish to clear all information on a previous program version, you have the possibility to do so when selecting your program files (by clicking on the *Delete Existing Units* button).

Only the code for the host and node programs and the "init" files are required for program execution on the target machine.

5.2 The User Interface

5.2.1 Motif

The interactive version of VFCS has a user interface which is based upon X Windows and OSF Motif.
The X Window System is a public domain network-based windowing system developed at the Massachusetts Institute of Technology. It has been implemented for UNIX, VMS, DOS, Macintosh and other operating systems. X is an appropriate basis for any kind of graphical user interface.

The popular MOTIF toolkit has been used to construct the VFCS user interface: MOTIF resides on top of the X Window System. MOTIF-based applications generally have fewer lines of code than those written using the pure X Window System objects, but require more memory.
Within the MOTIF environment, users generally use a mouse to point at and select an object with which to work. Selection is done by simply clicking the left mouse-button. In the descriptions below, we assume a very basic knowledge of working with window systems and a mouse. We give other details on using the menus and windows of VFCS explicitly.

5.2.2 Windows and Boxes in VFCS

We describe the main window, the file selection box, the unit box, the data distribution window and the command interpreter box in this section. There is also a brief description of the other boxes used in VFCS.

5.2.2.1 The VFCS Window

When you start up the system, the VFCS window with the title bar **Vienna Fortran Compilation System** will appear on your screen (see Fig. 5.4). The window is subdivided into two main areas. In the **program window** the user is able to see the current (possibly transformed) state of the chosen program unit and is able to select program statements (e.g. do-loop headers), in order to transform them or to request information from the system, with a mouse-click.

Below the program window is the **status window**, which tells the user what he or she (or the system) has already done and gives further information on the activities of the system. Some error messages are also printed here.

The **'Transformations'** button is located on the right-hand side of the status window. It is used to invoke the **Transformation Menu** (5.3.8). The **'Quit VFCS'** button beneath this may be used to quit the system at any time. Files will not necessarily be saved.
The **Main Menu** is displayed on the second line of the VFCS window at all times while the system is in use.

5.2.2.2 The Status Window

The lower part of the main window is used to display messages which inform the user about the actions of VFCS that are being or have been performed. At start-up time, this box has a default depth of 5 lines. Sometimes you may want to see more information about previous system operations. So that you can change the size of this window, there is a sash bar between the text window and the status box. Click on the small box located on this bar (to the right of the screen just below the Main Window) and move it upwards to enlarge the Status Window, or use the scrollbar on the right to see information lines which have been scrolled off the screen.

5.2.2.3 File Selection and the File Selector Box

This box (cf. Fig. 5.5) is used to search for and select the files which make up your source program. It contains a filter selection line, areas for displaying immediately accessible directories and files in the current directory, and a row of buttons: the *OK*, *FILTER*, *CANCEL* and *OPTIONS* buttons.

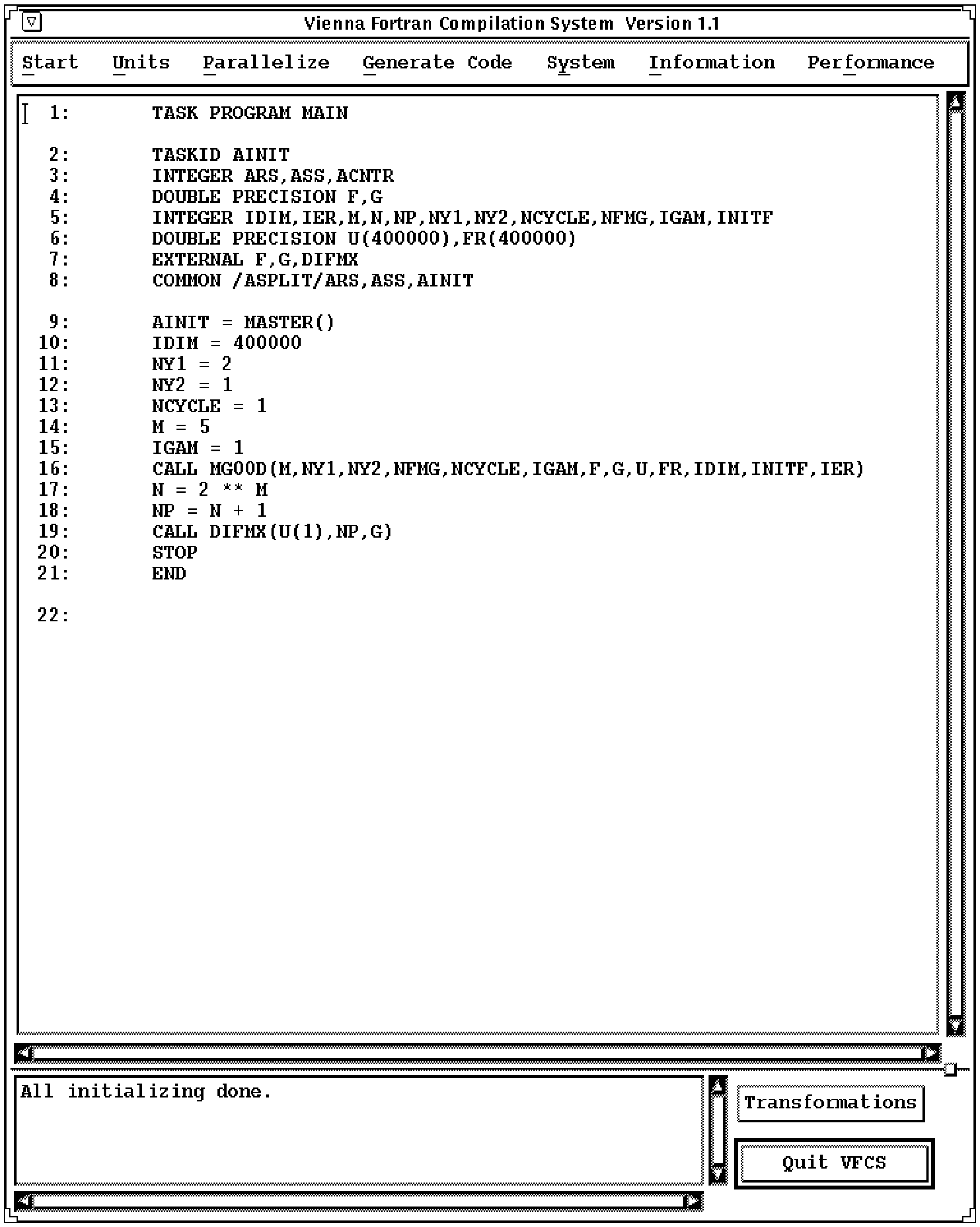


Figure 5.4: The VFCS window

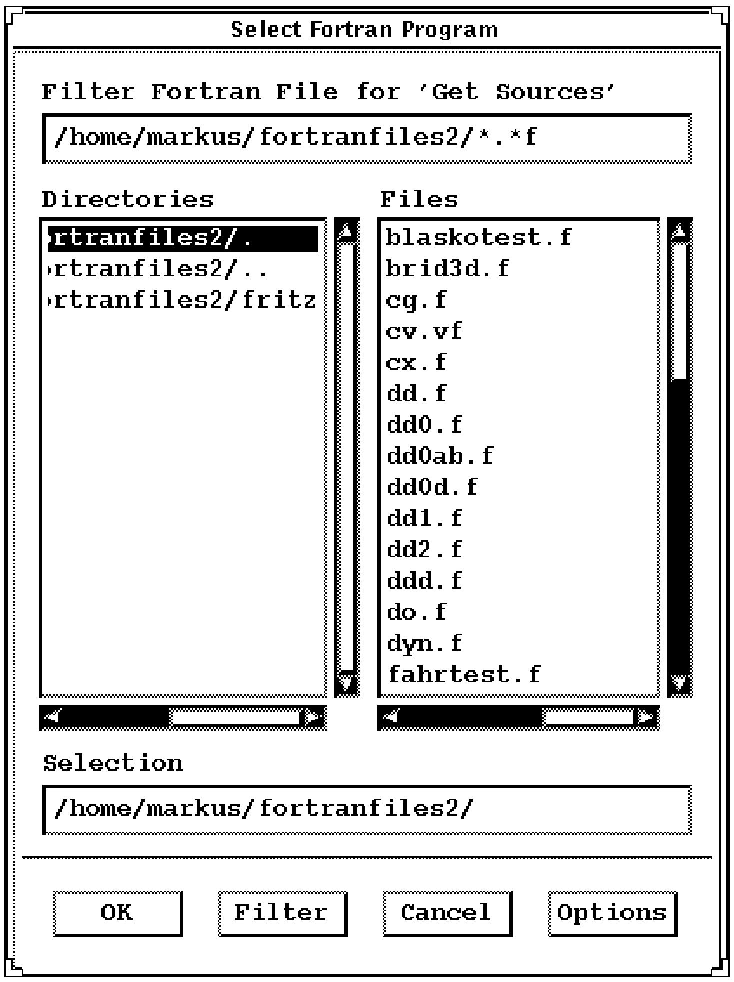


Figure 5.5: Select a Fortran program using the file selector box

The *filter selection line* is at the top of the display area. A standard filter is preset so that you are shown all files of the form "*.*f" within the currently selected directory. To modify this filter, click on the end of the line, delete unwanted characters using the *BACKSPACE* key and enter the filter you require. Click the mouse button again to see the corresponding file names.

You may **select files** anywhere in the local network that is accessible from your current directory. Double-click the entry **"<current_dir>/.."** to ascend into the next higher directory within the hierarchy. To descend into a subdirectory, double-click the appropriate entry in the directory list. To select a directory, you may also single-click the entry in the directory list and then the *Filter* button.

If your entire program is contained within a single file, you select it by either double-clicking the entry or single-clicking it and then press the *OK* button. Otherwise, you should select individual files by pressing and holding down the <CTRL> button and clicking the left mouse button. When you have selected all files for the program, press the *OK* button.

To leave the file-selector box without selecting files, press the *CANCEL* button.

The *OPTIONS* button is used to switch off one or more of the standard transformations which are normally automatically applied to the program in the initial phase. These transformations are described in Chapter 6. It should not be necessary to modify the standard setting for parallelization runs.

The *Delete Existing Units* bar will appear beneath the buttons if your current directory already contains internal VFCS data on program units: click on this to clear the system of all existing information on program units. This will ensure that you do not inadvertently use information derived from old versions of your code and enables you to prepare a directory for parallelizing a new program. It is not necessary to use this, however, if you replace old versions of program units by new ones, since the earlier information is cleared by the system in this case.

5.2.2.4 The Unit Selection Box

This is a special purpose box (cf. Fig. 5.6) to enable selection of one of the available node and host program units (main program or a subroutine or function). This not only displays the text of the unit in the main window; it also **activates** the selected unit so that lines or constructs may be selected from it for transformation or for the purpose of obtaining information about them. To select a unit, you either double-click the corresponding entry in the list, single-click the entry and press the *OK*-button in the same half of the window, or write the name of the unit into the selection-line and press the corresponding *OK*-button.

This box remains on the screen until it is explicitly cancelled. It typically remains open for the duration of a parallelization session, so that you have fast access to all parts of the program whenever you want to study the effects of previous operations. To remove it

- select the *Cancel* button to pop away the unit selection box.

This box may be selected and canceled at any time during the parallelization process.

5.2.2.5 The Data Distribution Window

This window contains two sets of buttons, an input line and a display area which enables you to view previous input lines (cf. Fig. 5.7). The top row of buttons have the following functions:

- **Batchread**: if you select this, you will be shown the file selector box so that you can specify a file containing data distribution specifications. They will be processed in silent mode: once the file has been read, the window disappears.

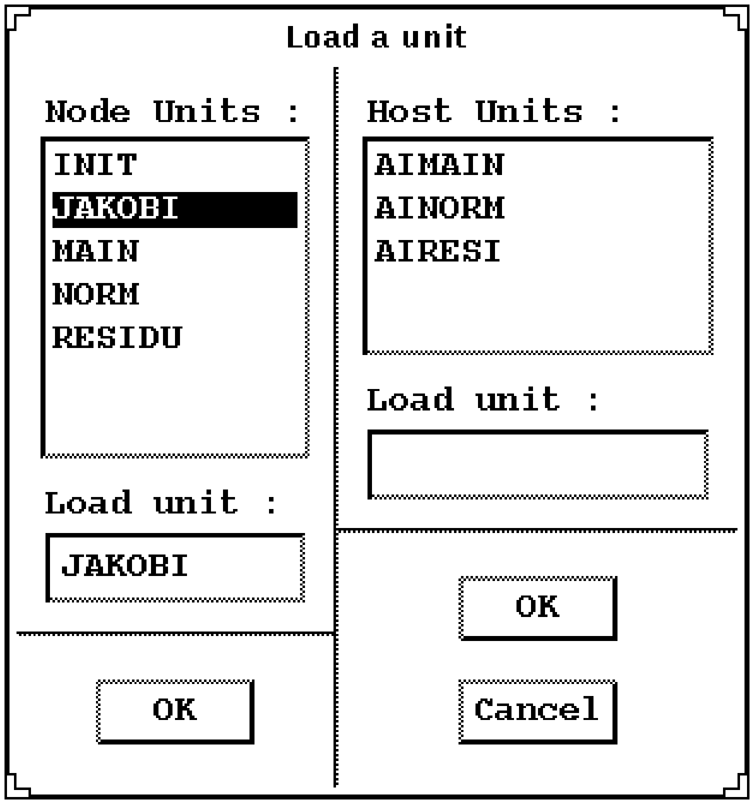


Figure 5.6: Select a program unit using the unit selection box

- **Read**: if you select this, you will be shown the file selector box so that you can specify a file containing data distribution specifications. They will be processed in prompt mode: each line of the file will appear on the input line so that you may confirm or modify it. To do so, click the *Execute* button in the lower part of the window. You may modify the text of the input line as required before confirming it. To cancel the line, click on the *Erase Line* button. Once a line has been processed, it will appear in the display area above the input line.
- **Save**: The distribution specifications you have input are saved to a file. The default name for this file is <program_name>.spec and it will be stored in your working directory. It may be subsequently read in by the *Read* or *Batchread* commands.
- **Change**: you may use this to view the current distribution and, if desired, to modify part or all of it.
- **Show**: you may use this to view information on the current data distribution. There are a number of options available. Arrays are classified according to their distribution and communication requirements; this internal classification is also shown.

You may specify distributions for arrays in the active unit by means of commands in a special data distribution language. The distributions it permits are described in Section 3.2 and the window and language commands are described in detail in Appendix A. Data distribution files are described in Section 3.2.5. Please refer to these sections for all further information.

Commands are entered manually via the input line or are read from a distribution file. You may write this file using the same language constructs as for interactive input. A unit must be activated before its local arrays may be distributed (via the *load* command). A command is not processed until you click on the *Execute* button. Some of these commands will result in the creation of further windows, in which you may view information, see the results of system analysis, or select a distribution from those derived by the system (if general block distributions are used with variables). In most cases, the use of these windows should be self-explanatory.

To leave this window, select the *Quit* button. Note that it is not possible to use any other parts of the system while this window is active.

5.2.2.6 The Command Interpreter Window

The functions provided within this window (see Fig. 5.8) enable you to record the actions taken during parallelization for subsequent automatic execution in batch mode. They also enable you to select batch macros, to execute them in silent and verbose modes, and to delete them. The row of buttons to the right of the window enable you to delete one or all available macros (*Clear* and *Clear All*), begin protocol activity in one of two different modes (*Prot* and *RProt*), switch off the protocol activity and write the protocol to file (*Proff*) and select a batch file for execution (*Batch*). If the batch file contains macros, the window will be extended to the right to display a list of these. A macro may be selected for execution from this extension.
For all details of this facility, including the batch command language, see Chapter 7 and Appendix B.

5.2.2.7 Other Boxes

The *View Units* box presents you with a list of unit names from which you may select a unit, for viewing only, by clicking on the name and clicking the *OK* button, or by double-clicking the name. Leave the text window by clicking *CANCEL*.

Many of the items in the *Information* menu present details of the program by means of pop-up boxes which you may enlarge if the information does not fit into the standard size. Scroll bars are available below and to the right of display windows for easy viewing of the information.

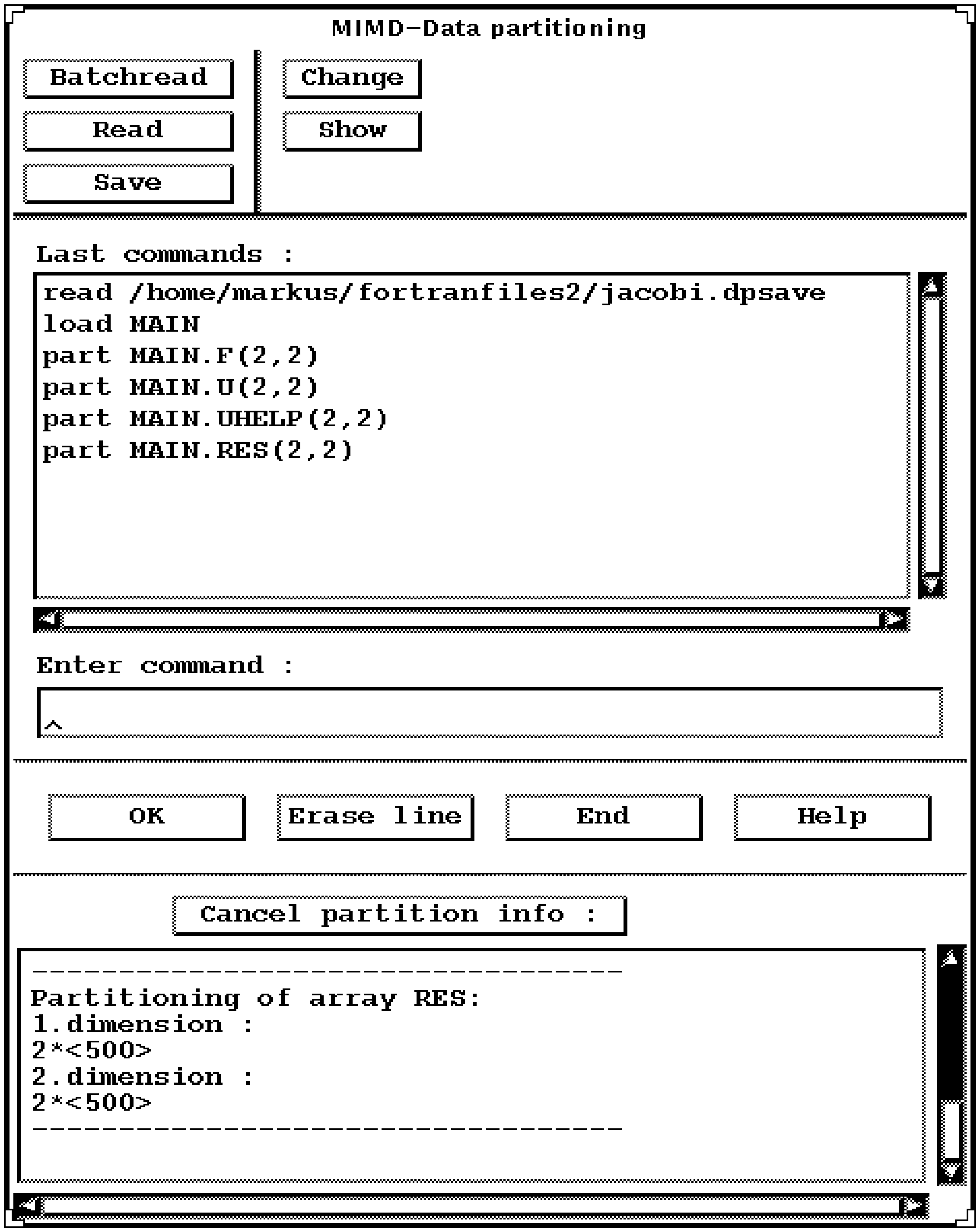


Figure 5.7: Distribute arrays using the data distribution window

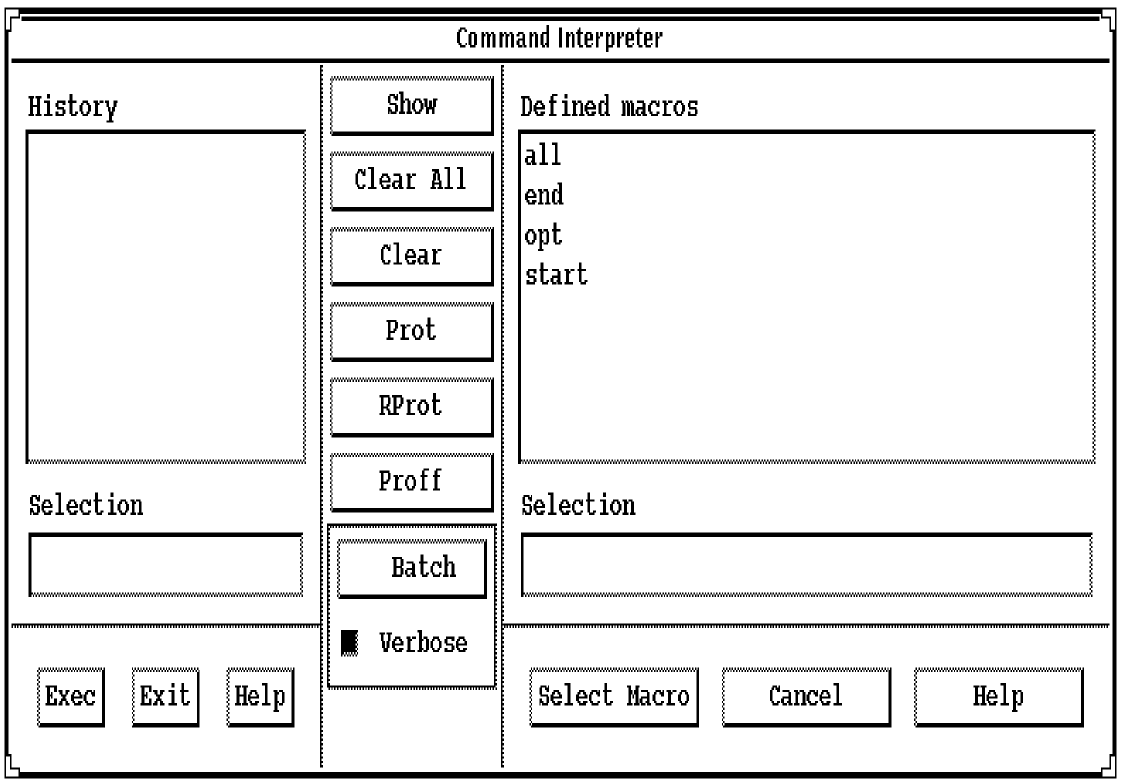


Figure 5.8: Command Interpeter Window

Pop-up boxes are also used by the system to display error messages. They must be explicitly cancelled by the user before work can proceed.

5.2.3 Important Functions and Options

5.2.3.1 Selecting Menu Items

In VFCS, you may select a menu by pointing at it with the mouse and then clicking the left mouse-button. Point at a submenu or menu item and then click again to select it.
The menus are also selectable from the keyboard via so called mnemonics: these are abbreviations of the pulldown-menu-text. All mnemonics consist of one character only; they are underlined in the menu entries.
To select a top level menu, press and hold down the Meta-key and press the mnemonic character. To enter the *Start* menu, for example, hold down the Meta-key and press **'s'**. This is the underlined mnemonic for the *Start* menu. To select menu items within *Start* and other items in second level menus, simply press the underlined mnemonic-key without any additional Meta-key.
To leave a menu without making a selection, click the mouse button anywhere outside the window or press the *ESC* key on your keyboard.

Menu items which are faded are disabled: they are not available for selection by the user and clicking on them will have no effect. In most cases, this mechanism will prevent the user from applying transformations to a program in an illegal order.

5.2.3.2 Selecting a Line or DO-Loop

Some transformations and menu items require that you select a line or DO-loop before they can be applied (or during their application). Some require more than one line or DO-loop. Some kinds of information are only available for a line of code or for a data structure in a particular line.
Single-click the line you require anywhere within the line to identify it to the system. To select a DO-loop, single-click the DO-loop header anywhere within that line.
If you have not already selected the line you require before you click on a transformation, you will be prompted to do so. Single-click the line as above to select it. The prompt box will generally disappear without further user interaction. If another line or loop must be specified, you will again be prompted to specify one. Clicking on the *OK* button without clicking on a line will mean that the transformation cannot be carried out. Click on the *CANCEL* button to quit without making a selection.
To reselect the same line for another purpose, single-click another position within the line.

ATTENTION: MOTIF does not permit you to single-click the same line at the same cursor position twice.

5.2.3.3 Selecting a Subroutine or Function Call

If you select a particular subroutine or function for *Inline Expansion*, you have to double-click the first character of the routine's name within the line of code where the call appears.

5.2.3.4 Providing Information within a Box

Some pop-up boxes will prompt you to provide the system with some kind of information. If a pop-up box requires you to specify a line or DO-loop, do so as described above. If other information is required, an input window will be made available for you to type in the appropriate data. Confirm your input by clicking on the *OK* button; cancel the operation by clicking on *CANCEL*.

5.2.3.5 Leaving a Box

In general, pop-up boxes must be explicitly removed from the screen: further operations are suspended until this is done. If no user interaction is required, then you should generally press the *CANCEL* button to remove the box. If you are asked to confirm some input or item of information, then you should do so by pressing *OK*; *CANCEL* is used to exit a box without making a confirmation.

Both the *Unit Selection box* and the *Transformation Menu* (which – in contrast to all other menus – is really a selector box, and remains on the screen until it is explicitly cancelled by the user) do not suspend further operations while on the screen.

5.2.3.6 Help

The Help menu items have not been included in the current version of VFCS.

5.2.3.7 Focus Lock

Normal MOTIF-behavior includes a focus-lock whenever a pulldown-menu is on the screen. This means that all keyboard and mouse input is locked until a submenu has been selected or the pulldown-menu is canceled. This is normal MOTIF-behavior. It means that, for example, you cannot input any text (e.g. for text-editors) on the screen, move other windows, or take snapshots.
If for some reason this is inacceptable behavior for you, you should switch the *Special Focus* item of the *System Options* submenu in the *System menu* to **'On'**. This setting will remain in effect for subsequent sessions if you leave VFCS with a normal *Quit*.

5.2.3.8 Other System Options

There are several toggle switches in the *System Options* submenu of the *System* menu to enable you to customize your environment. Current options permit you to automatically load the main program of your node program once source files have been read in, and to automatically display the *Unit Selection Box*.

5.2.3.9 The Newvfcs-Resource-File

This file contains information about the geometry and the colors of VFCS. It should be located in the HOME-directory. Do not change this file unless you know what you are doing !!

5.3 The VFCS Menus

During the parallelization of a source program, the user selects services and transformations in a hierarchical menu system. We describe the functions in the VFCS pull-down menus in the following subsections. Note that not all menu items are available for selection at a given stage of parallelization.

Here and in the following chapters, we use the following notation to specify a menu item:

[*main*-**menu :** *sub*-**menu : . . . :** sub^n-**menu**]

Example:
[**Transformations : Induction Variable**] refers to the **Induction Variable** item in the **Transformations** menu. To select it, you must enter the *Transformations* menu and then select the item *Induction Variable*.

5.3.1 Main Menu

The main menu, which is always presented in the second line of the screen, has six different entries (each of which is a pull-down menu) for different classes of services:

Start :
This menu entry includes all the services which are required before the actual parallelization transformation is applied to a program. This includes selection of the source program files, and the interactive addition or modification of Vienna Fortran constructs to enable data parallel program transformation. It is also possible to quit VFCS from inside this menu (5.3.2).

Units :
Once the source program files have been selected and the file selector box has been closed, the individual program units (remember, there are separate host and node units) can be accessed. There are entries in this menu for selecting a unit, which makes it the active unit, saving a unit, viewing a unit, printing a unit and deleting all existing units (5.3.3).

Parallelize :
The operations performed to modify the node program so that it will execute in parallel are invoked from this menu. There are entries for creating the defining parallel program, and for optimizing the parallel code. It is also possible to modify the work distribution and the communication in certain cases. Vector code may be generated.

Generate Code :
With this menu, the user can generate source code for one of the supported target architectures (Intel IPSC, Suprenum or Parmacs). It is also possible to insert timing instructions into the code to obtain performance information. The generated code is also written to file from this menu (5.3.5).

System :
The system menu provides a variety of functions to support the VFCS user. It includes a protocol facility, which will record all transformations applied during a parallelization session (cf. 7). It also contains utilities for managing different versions of a program. Various system options can be set.

Information:

This menu provides access to information about the program. Data-flow information, information about data dependences and overlap information will be shown. The user can examine the call graph and the communication in the program (5.3.7).

Transformations :

The transformation menu is invoked by clicking the button at the bottom right of the VFCS window. All user-selectable transformations for parallelization and vectorization are activated via this menu. If this menu is selected, it remains on the screen next to the VFCS window until it is explicitly canceled by the user. Thus it is possible to apply several transformations, or to examine each of the program units in turn, without having to leave and re-enter this menu each time. Most transformations can be applied to the program at any stage of parallelization before code generation. A few of them can also be applied after code is generated (5.3.8).

5.3.2 Start Menu

This menu (cf. Fig. 5.9) provides the user with following options:

Get Sources for Parallelization:
This is the usual place to start parallelization of source programs. Once they have been selected as described in 5.2.2.3, the frontend is run automatically. It performs scanning, parsing, and normalization and standardization transformations (cf. 6.1). Some global information about the program, e.g. the call graph, will be computed. In addition, the source program is split into host and node units (4.2.2).

Specify Data Distribution:
This is the menu item which invokes the data distribution window (6.4.1.1 and Appendix A). Within it, the user can specify a data distribution which is the starting point for the subsequent parallelization. An existing data distribution may be displayed and modified from this window also.

Mark Parallel-Loops:
Do-loops can be marked by the user for transformation into Forall loops (6.6). This is useful if the system is not able to recognize that a loop is parallel. The loop is selected as described in 5.2.3.2.

Get Sources for Sequential Work:
This is used to get the source program in the same way as in the corresponding menu entry for parallelization at the top of the *Start* menu. Again, the frontend will run automatically, performing scanning, parsing, and a set of transformations, and computing the call graph. **In this case, however, the program will not be split into a host and node program!** The program cannot be subsequently parallelized, except by reselecting the sources in the corresponding menu for parallelization. This permits you to apply transformations to your sequential program, including generating vector statements.

Quit VFCS:
You can quit VFCS with this menu entry or by clicking the *Quit* button. There is no difference in functionality between the two. If you have changed any system options, they will be saved and the same settings will be available the next time you start VFCS.

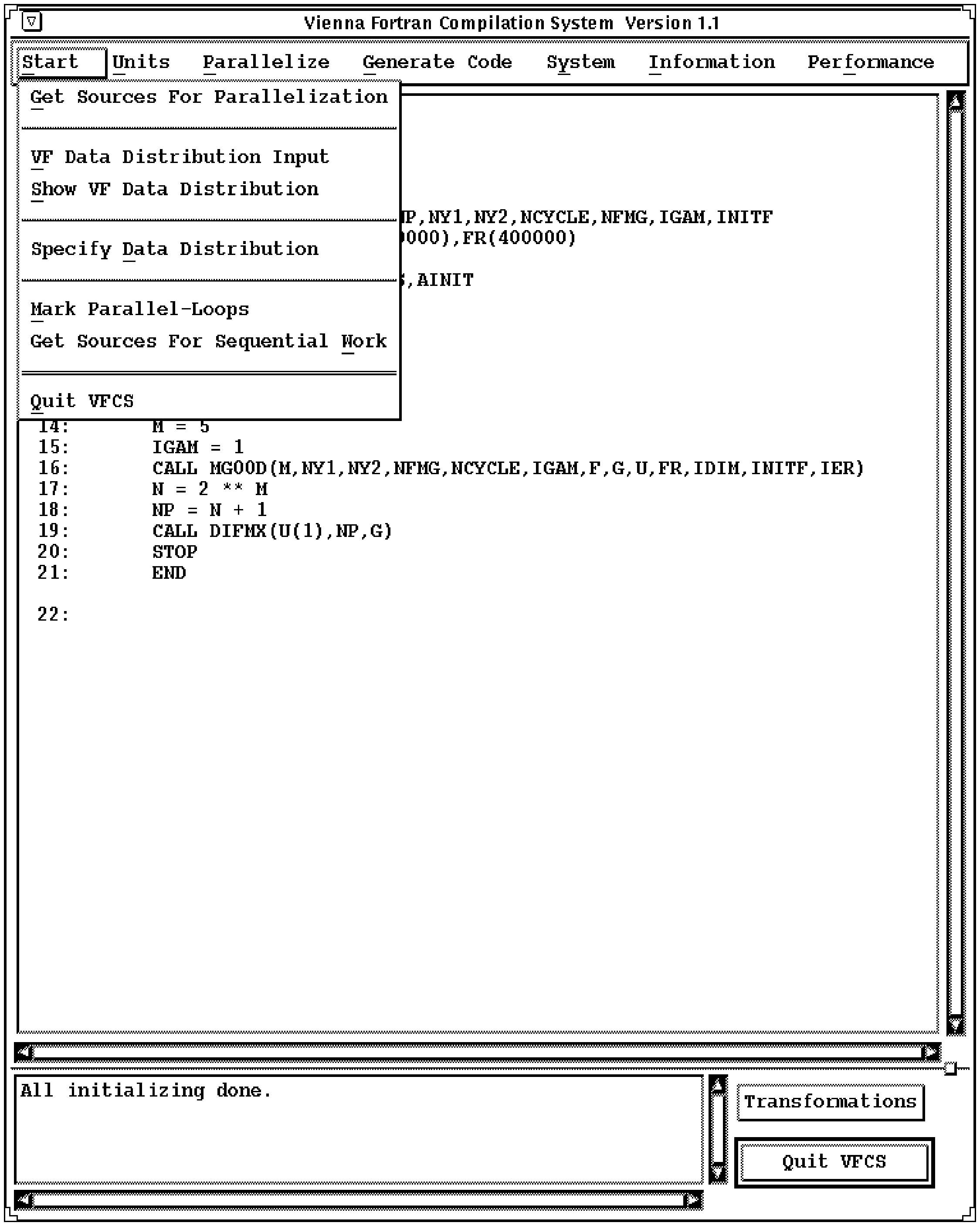


Figure 5.9: Start Menu

5.3.3 Units Menu

Select Unit:

This is used to call up the *Unit Selection Box* (5.2.2.4), which displays the list of host and node program units and permits one of them to be selected (cf. Fig. 5.10). The selected unit will be loaded into the program window and thus activated. The user may interactively define, view or modify the distribution of data within the active unit, or view and apply transformations to code within it. Information specific to the active unit is available from the *Information* menu. The *Unit Selection Box* is canceled via the *Cancel* button.

Save Unit:

The *Select a unit for file output* window is displayed. The user can choose a program unit from the list or write a unit name directly into the *Selection* box. It is then written to a file. In the current version, this file is always given the name "fortran", which must be modified by the user to reflect its contents.

View Unit:

Here, too, the user must select a program unit from a list presented by the system. The selected unit is shown in the *Text from unit* window. It is not active, however, and thus cannot be modified. Lines cannot be selected to obtain system information from this window. Leave this window via the *Cancel* button.

Print Unit:

The user is again provided with a list of units from which one should be selected for printer output. With *Change Printer* the user can specify a printer other than the local default.

Delete All Units:

This may be called to clean the subdirectories of all internally constructed information files before a new session is begun. Your source program will not be deleted; you may begin an entirely new session on the program you have worked with previously, or on a new program. Any versions you have set will be lost.

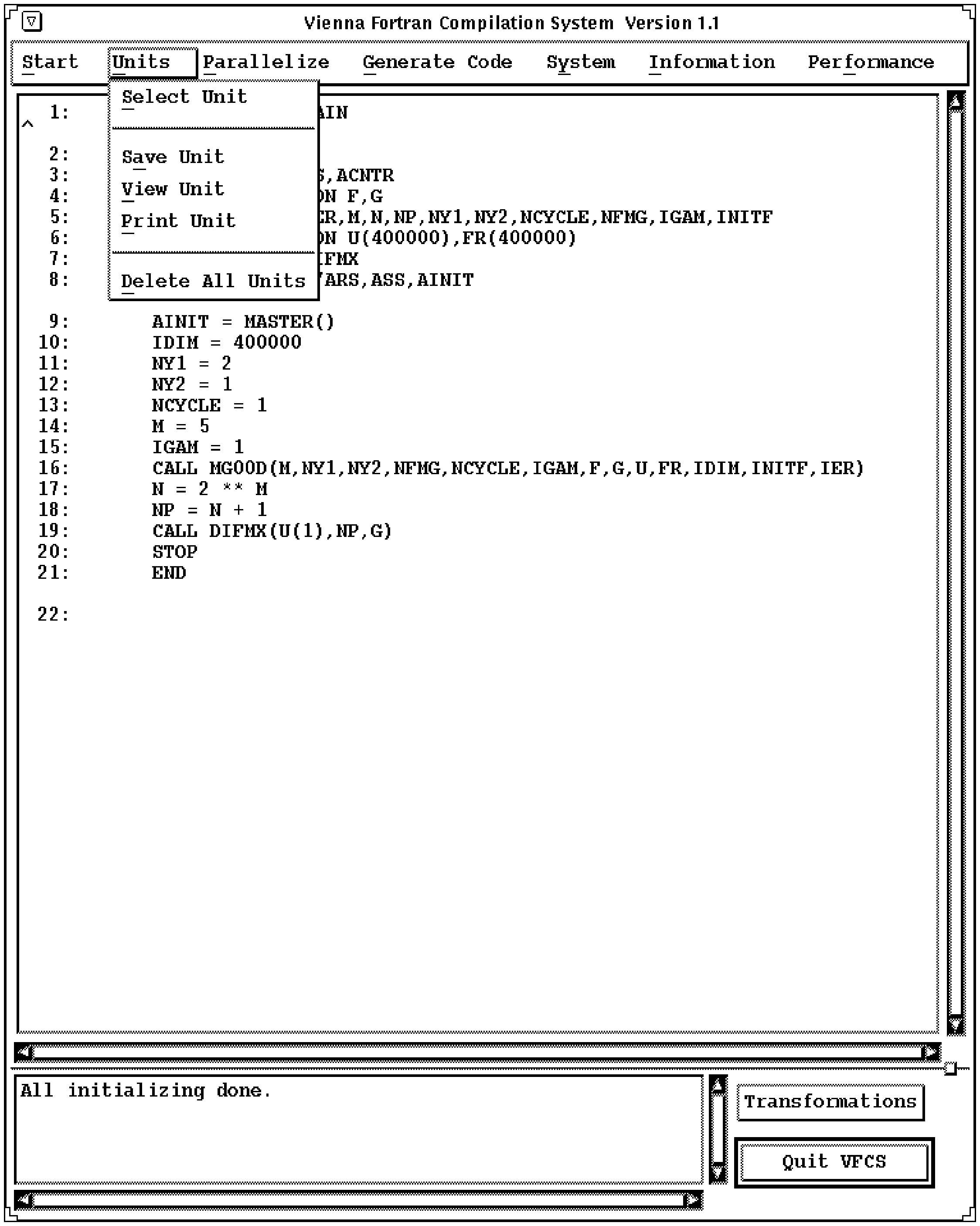


Figure 5.10: Select a program unit

5.3.4 Parallelize Menu

This menue and submenus are discribed in Fig. 5.11

Start Parallelization:
This transformation creates the defining parallel program according to the specified data distribution (see 4.3). VFCS has an internal notation to describe the results of this program transformation for the user to see (in 5.1.6).

Optimize Communication:
The communication is optimized in all units. This optimization transformation extracts the communication from surrounding loops wherever possible (4.4.4).

Optimize Masking:
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" (or guards) by an appropriate adaptation of the loop bounds in the internal representation (4.4.5).

Modify Work Distribution:
The following items for work distribution (cf. Fig. 5.12) can only be used in conjunction with FORALL loops. (For more details, see 6.6).

Cyclic Work Distribution:
Generates a cyclic work distribution: loop iterations are distributed to the processor in a cyclic fashion.

Block Work Distribution:
Generates a block work distribution: loop iterations are distributed to the processor in a block fashion.

Work Distribution By Owner:
Generates a distribution whereby each loop iteration is executed on the processor which owns a particular array element.

Work Distribution By Value:
Generates a work distribution according to a user-defined array: the *ith* loop iteration is executed on the processor whose linear number is equal to the contents of the ith element of the array.

Automatic Work Distribution:
The system generates a work distribution automatically.

Mask Propagation:
In contrast to the other transformations, this is performed within the active unit only. The masks (guards) of statements may be propagated to other statements in the same loop nest; this transformation may eliminate the updating of replicated variables(6.4.1.2).

Create Vector Code:
This transformation transforms FORTRAN 77 DO loops into Fortran 90 array statements. This may be useful for the transformation of sequential code or for appropriately transforming the node program to execute on machines with a vector unit.

Communication:
This item can only be selected if you are parallelizing a FORTRAN 77 program. It is a submenu with two entries:

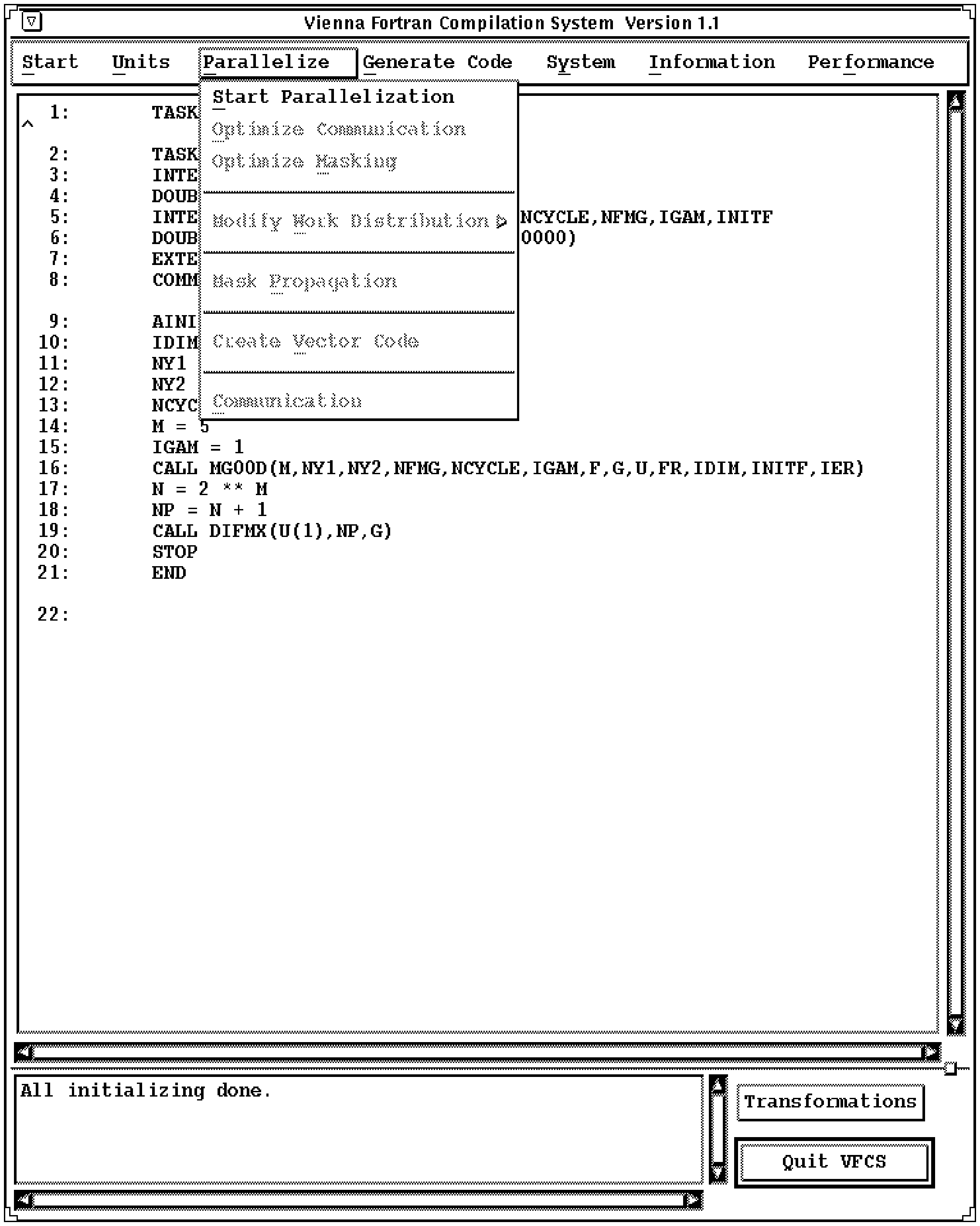


Figure 5.11: Parallelization Menu

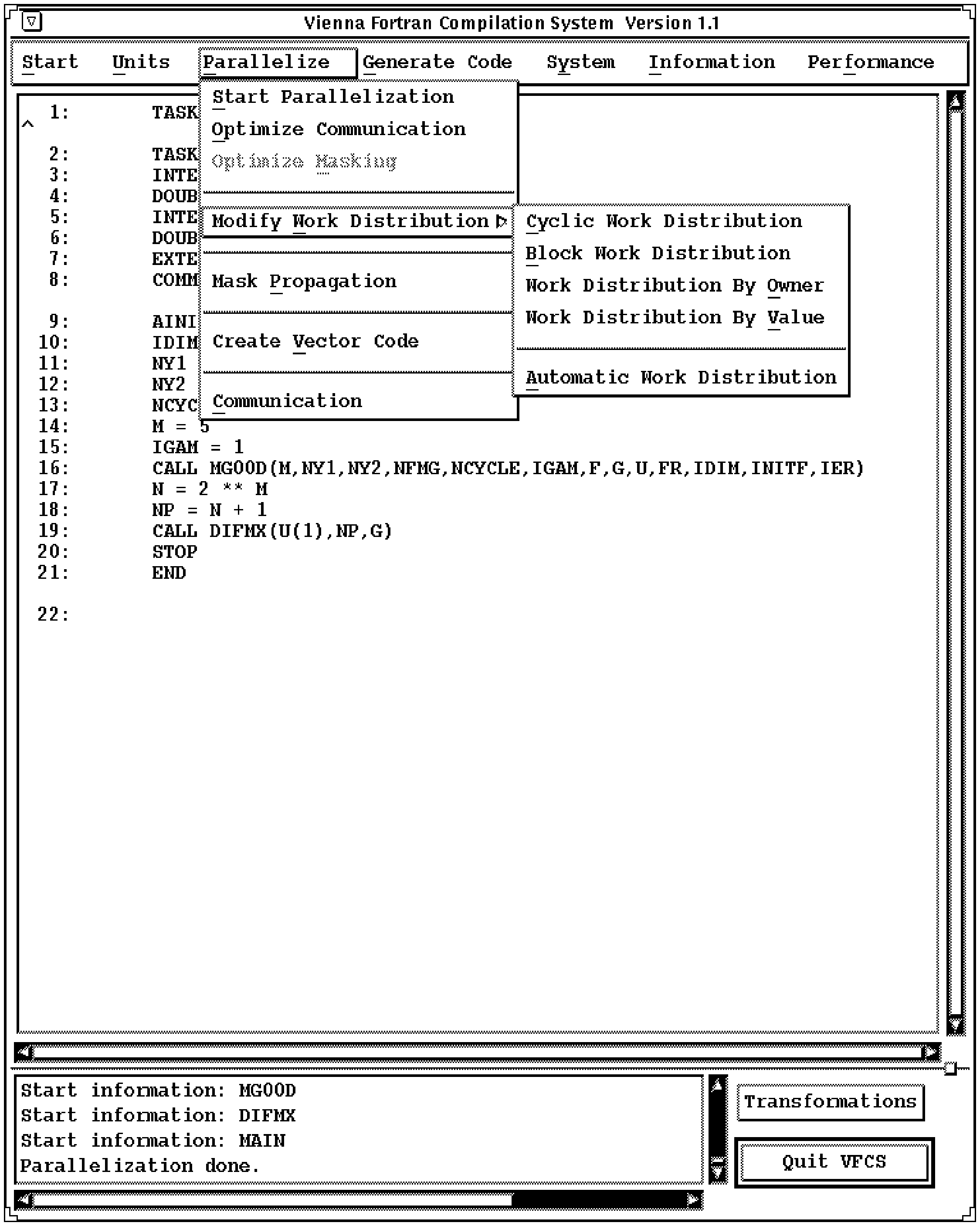


Figure 5.12: Modify Work Distribution

Show:

This information is statement based: it displays information on the parts of an array which will be communicated from one processor to one or more others by a particular statement in the program.

Change:

The system will not always be able to detect that a particular communication is superfluous. This provides the user with an "escape hatch" to modify the system-generated communication statements. It is to be used with extreme care and only by an experienced user, since the inadvertant deletion of necessary communication statements may cause the program to produce erroneous results.

5.3.5 Generate Code

Select Target Machine:

This menu (see Fig. 5.13) is used to specify the target machine you want to generate code for. It then automatically adapts the parallel program for the selected architecture. Code can currently be generated for several different target machines:

Ipsc:

If you select this item, the VFCS generates code for the Intel iPSC/860 machine.

Ipsc With Communication Protocol:

Code is again generated for the Intel iPSC/860 machine. However, the execution of SEND and RECEIVE statements are traced: the contents of the messages are protocolled.

Suprenum:

Code is generated for the Suprenum machine.

PARMACS:

Code is generated for machines which support the PARMACS (Argonne/GMD Macros for Portable Parallel Programming), Version 5.0 or 5.1.

Insert Timing:

This menu (see Fig. 5.14) consists of the following items which permit the insertion of timing constructs for individual regions of the program (which may be as small as one statement):

Mark Time Measurement:

The user selects the region for which a simple measurement of execution time is required. This must be done **after** the last *Generate Code* phase. When this menu item is chosen, the user is asked to mark the first statement and the last statement of the region to be measured. Marking of code regions has to be done in all units separately. Selections may be removed again by selecting the submenu *Delete Time Marks*.

Insert Time Procedures:

After all regions have been marked for timing as desired, this submenu **must** be chosen. It will insert the following statements in the program:

```
TIME = DCLOCK()   {* before marked block *}
CALL TIMINF(time-DCLOCK(),unit_name,
                 first_line,last_line)
                  {* after marked block *}
```

During execution, subroutine TIMINF collects information about the process, unit name, number of the first and last statement of the block, and execution time in seconds. Procedure TIMINF is included in the VFCS runtime library.

Delete Time Marks:

The selected region will be unselected, so long as the menu item *Insert Time Procedures* has not been invoked. (It does not remove inserted time procedures.)

Save Parallel Program:

The parallel code is generated in source form from the intermediate representation. It is saved in host and node program files with the appropriate syntax for the selected machine. These files can be inspected by the user (see 5.1.7 and 8).

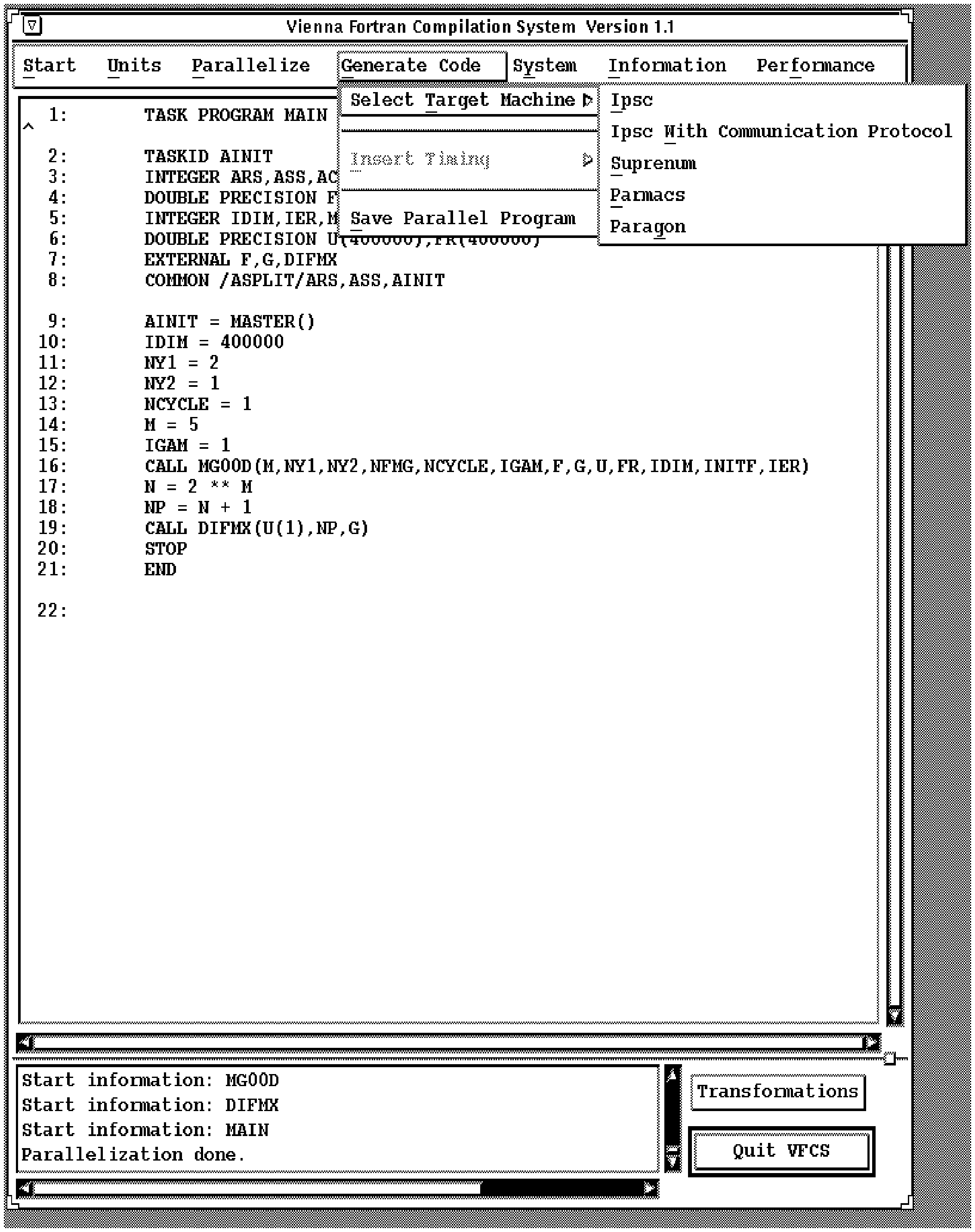


Figure 5.13: Generate Code for different target machines

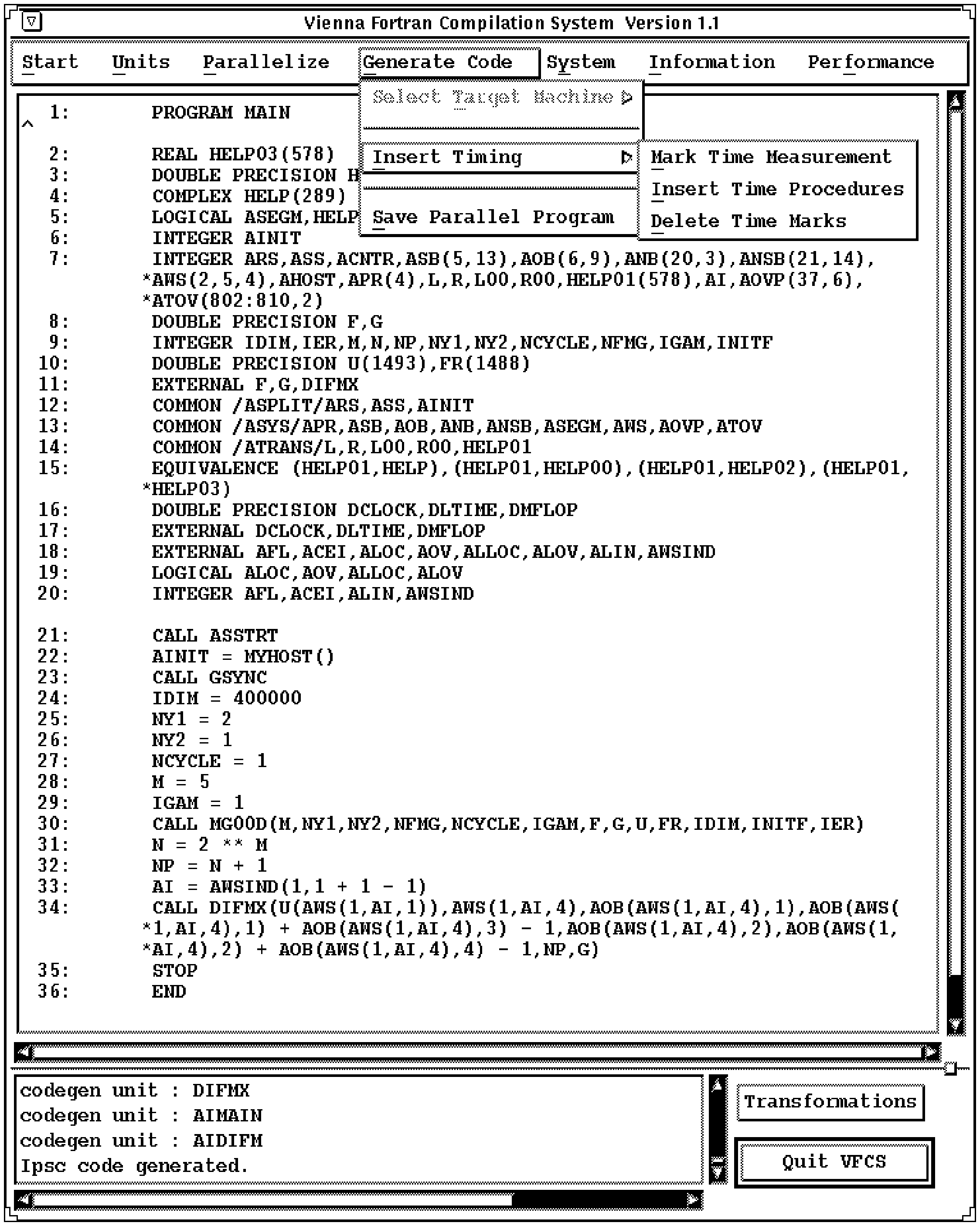


Figure 5.14: Insert timing statements for profiling

5.3.6 System Menu

This menu (cf. Fig. 5.15) provides the user with following options:

Batch Facility:
This calls up the command interpreter window, which enables the creation, deletion and use of batch command files (see 5.2.2.6 and Chapter 7).

Set Version:
The current state of the parallel program is saved to file under the name given by the user. The parallelization process can be restarted at that position by restoring the version. This is also possible during a subsequent session, since program versions are saved until explicitly deleted 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.

Show State:
Presents some general information of the global system state of VFCS in your information window.

Goto Line:
You will be asked to specify a source line. The system will select this line in the active unit.

Search String:
Opens a dialog box to enable you to search for text patterns. Enter the desired text into the search-pattern text input field. The following options are provided:

- **Find next:**
  Look for the next occurrence of the specified text. Each occurrence is highlighted. The text scrolls to the occurrence.
- **Show all:**
  Show all occurrences of the specified text.The selected text window scrolls to the last occurrence.
- **Clear all (time-) marks:**
  Unhighlights all marked positions in the text. Any highlighted time-marks (from timing options in the Generate Code menu) are unhighlighted too.
- **Select a text window:**
  This is used if you want to select a text string in any window other than the main window; for example, you may wish to find all EXSR statements in a program unit in second window opened by the View Units menu item. When you press this, the cursor changes into a pointing hand.You now may select a text window. The main text window is selected as default text.
- **Cancel:**
  Pops away the 'Search for string' dialog.

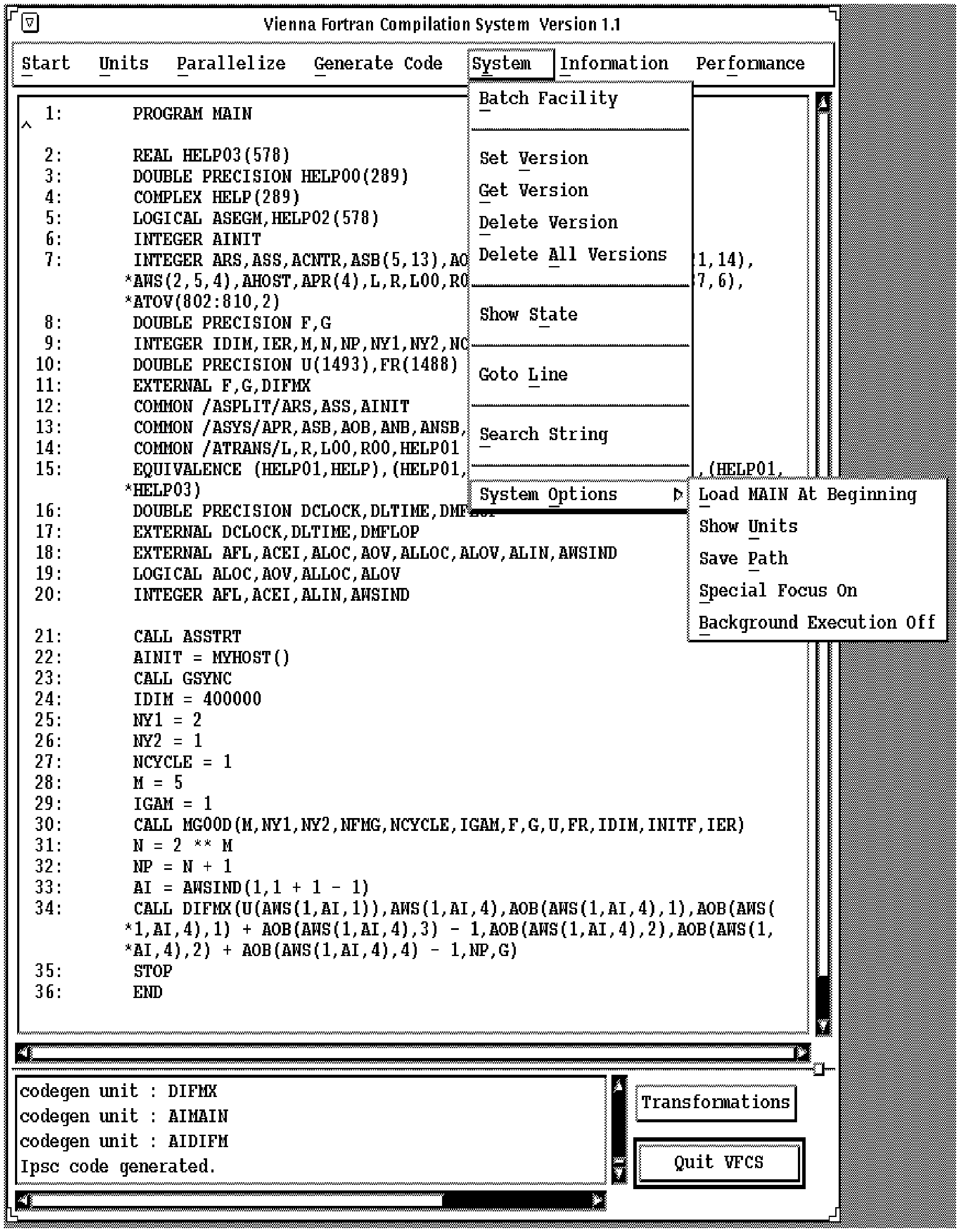


Figure 5.15: System menu

System Options:

With this submenu the user can change the following system options, which will be retained and take effect the next time VFCS is invoked.

Load MAIN At Beginning:

After starting VFCS the unit **MAIN** will be activated and appears in your program window.

Show Units:

The next time you start VFCS, the **unit selection box** will be automatically opened. No unit is automatically selected.

Save Path:

Save the current path for your frontend, spec file (containing a data distribution specification) and batch command file.

Special Focus On:

If *Special Focus* is on, you can use other windows, even if a VFCS pull-down-menu is open. This is a non-standard MOTIF feature which is not accepted by all window managers (5.2.3.7).

Background Execution On:

If *Background Execution* is **On**, other analyzing tools may be used in the background.

5.3.7 Information Menu

Data Flow:

You will be prompted to select a statement of the active program unit. Data flow information is displayed in a pop-up box (see Fig. 5.16) once you have selected the kind of information you require (6.5.1).

All Chains:

All of the data flow chains described in the menu items below will be shown for all scalar variables occurring in the selected statement.

UD Chains:

For each scalar variable in the selected statement, this shows all definitions which reach this statement (use-definition chains).

DU/True Chains:

If a scalar variable is defined in the selected statement, the uses reached by this definition will be shown (definition-use chains).

Output Chains:

Shows the output dependences for all scalar variables which occur in the selected statement.

Anti Chains:

Shows the anti dependences for all scalar variables used in the selected statement.

Use/Def Sets:

The use-definition information for the active program unit is displayed.

Data Dependences:

You should first select an item from this menu (cf. Fig. 5.17) to indicate which kinds of data dependences are to be displayed. You will then be prompted to select a statement or DO-loop for which you want to see the dependences (6.5.2).

All Dependences:

Shows all dependences for all variables occurring in the selected region.

True Dependences:

Shows the true dependences for all variables defined in the selected region.

Anti Dependences:

Shows the anti dependences for all variables used in the selected region.

Output Dependences:

Shows the output dependences for all variables defined in the selected region.

Show Overlap:

This is only available after the *Start Parallelize* menu item has been selected. Further, the user must ensure that the appropriate analysis has been performed by selecting the *Overlap Unit* or *Overlap All Units* items at the bottom of this menu (see Fig. 5.18).

Overlap In Unit:

Shows the computed overlap information in the active unit for the array specified. The maximum overlap for each distributed array may be displayed, or all overlaps in the active unit shown, by clicking on the appropriate button.

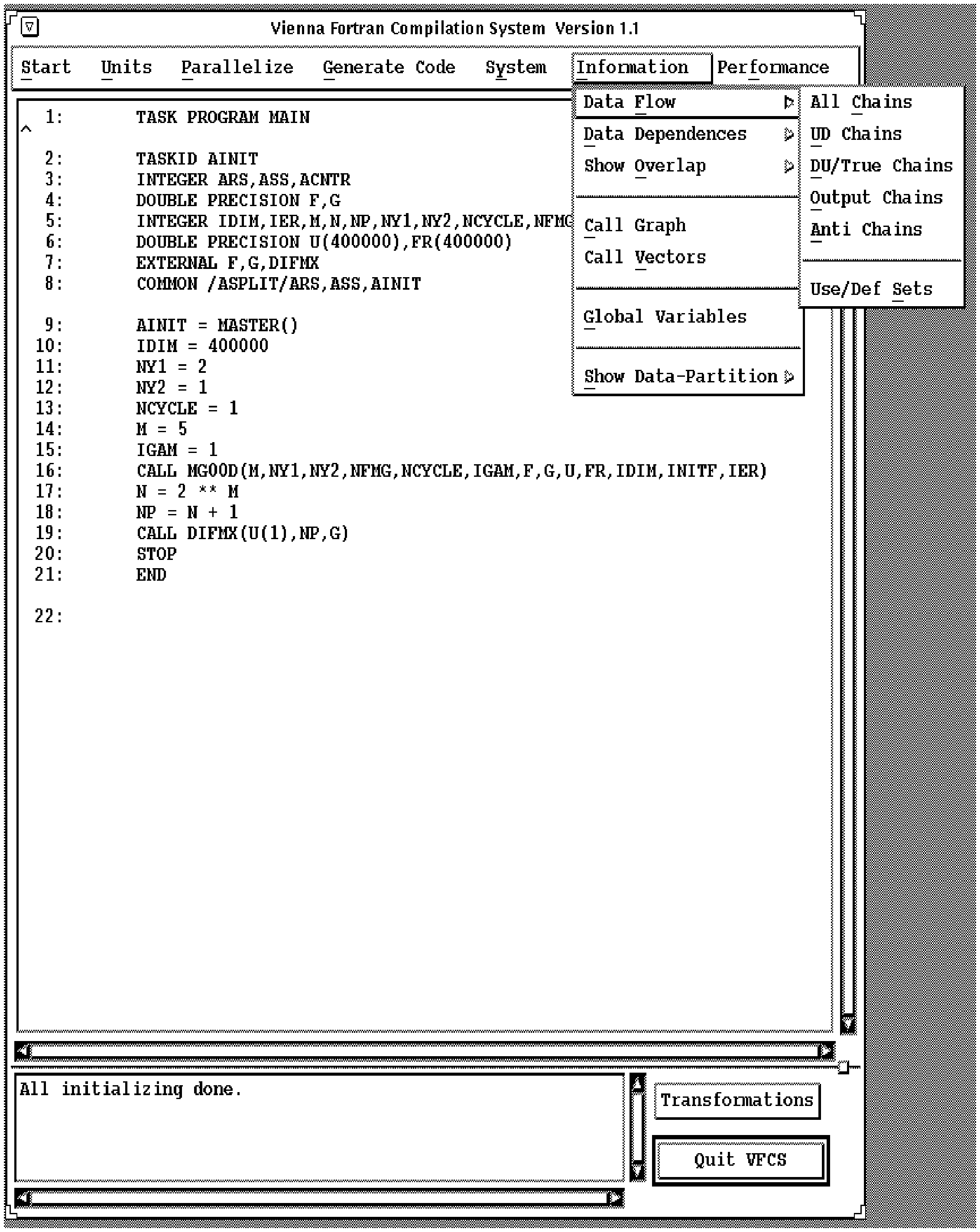


Figure 5.16: Data flow information

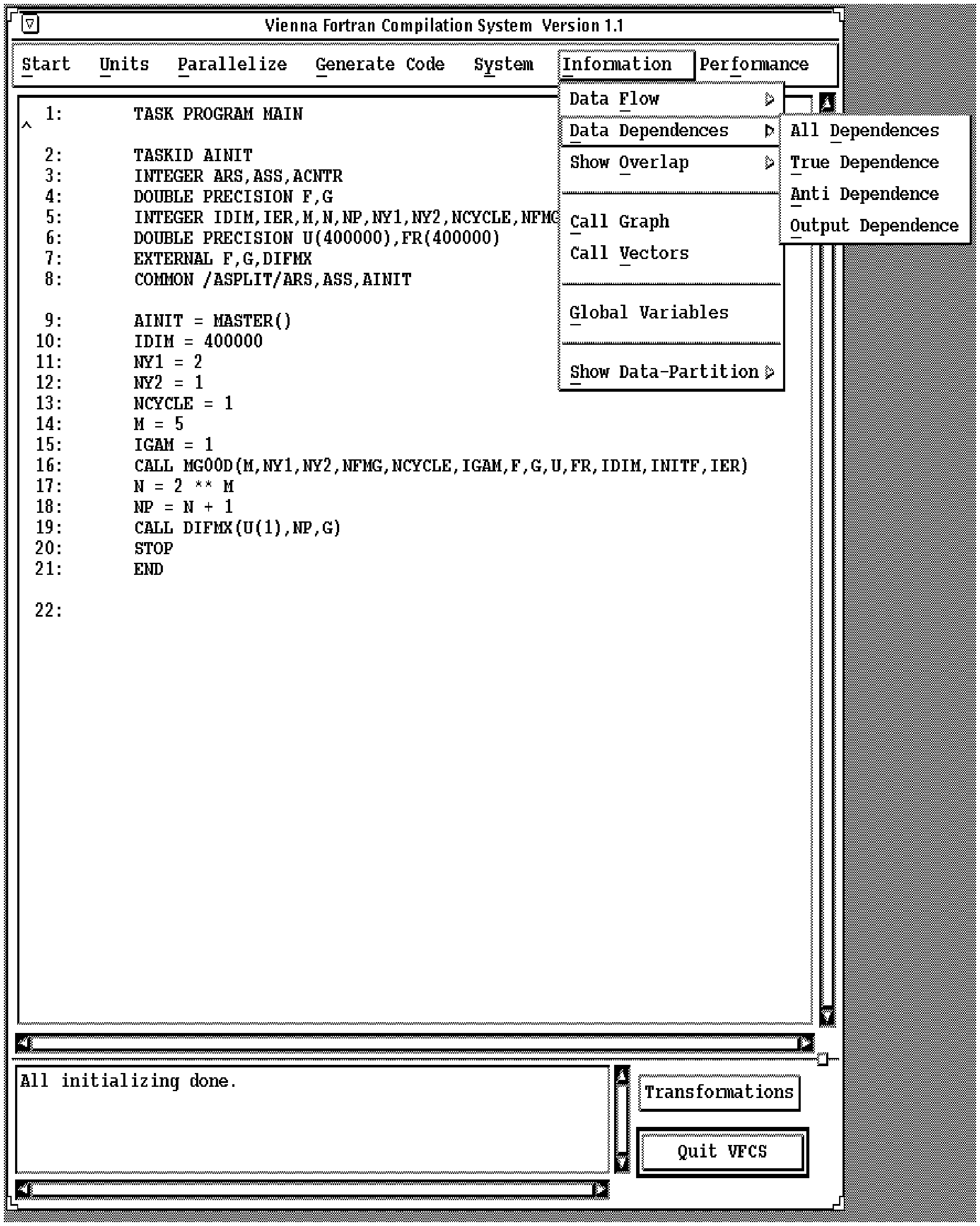


Figure 5.17: Data dependence information

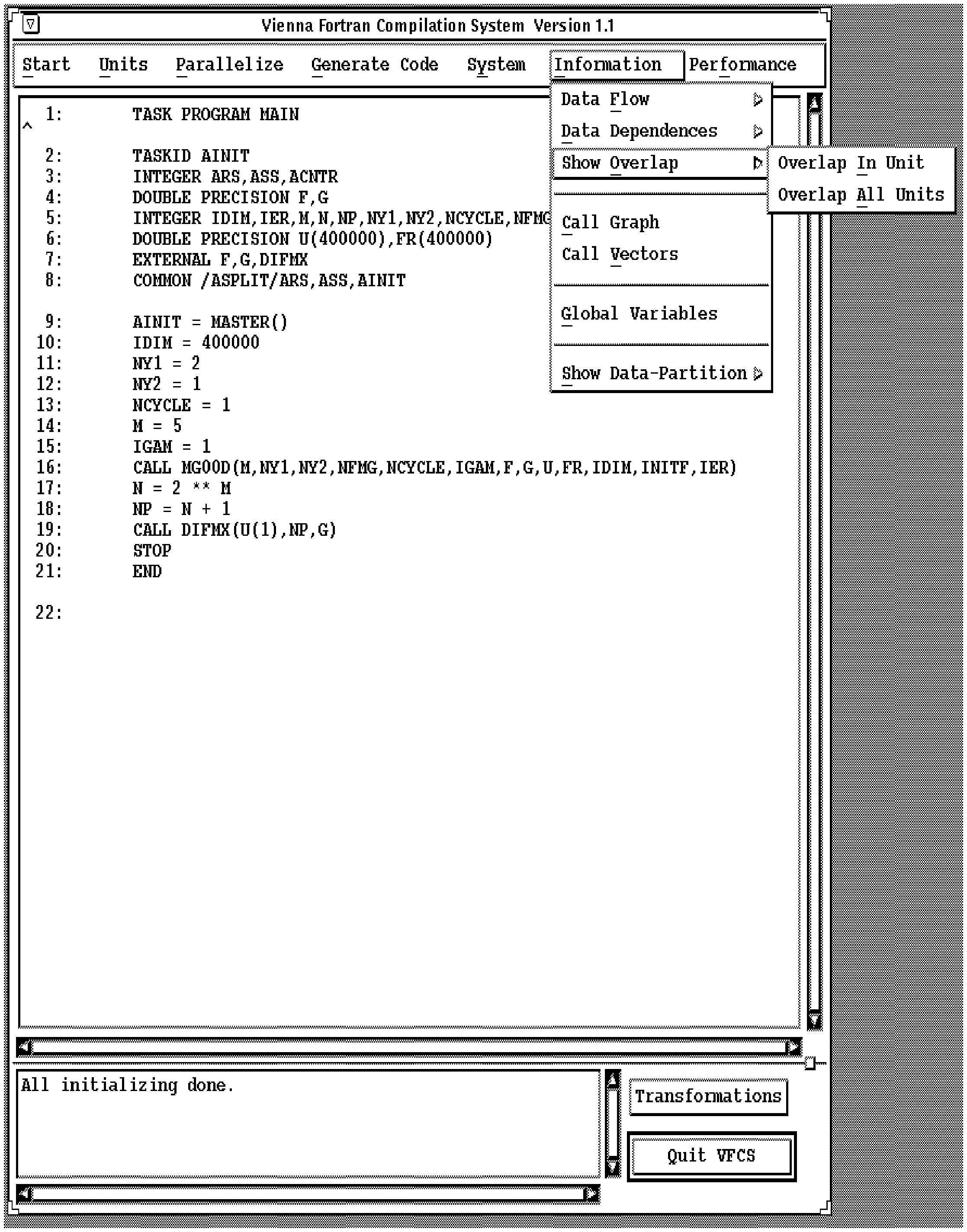


Figure 5.18: Show overlay information

Overlap All Units:
The computed overlap information in all units is shown. This global information is determined for individual units and for the entire program. It covers the entire overlap areas of the distributed arrays.

Callgraph:
A graphical representation of the call graph is presented.

Call Vectors:
All call vectors for all units will be computed and shown.

Global Variables:
This displays the global variables (dummy arguments and COMMON block variables) read or written in the active unit.

Overlap Unit:
This menu entry initiates an analysis of the overlap areas resulting from accesses to distributed arrays in the active unit. (See *Show Overlap* above).

Overlap All Units:
Computes the overlap information in all units. The computed overlap information in all units is collected and combined to derive global overlap information. This global information is determined for individual units and the entire program. It is made up of the entire overlap areas of the partitioned arrays. (See *Show Overlap* above).

5.3.8 Transformations Menu

All interactive transformations provided by the system are to be found in this menu (see Fig. 5.19). For detail information on the transformations, see Chapter 6. This menu, unlike the others, does not disappear until you explicitly *Cancel* it. Thus you may examine each of your program units, or perform further parallelization steps and leave this menu on the screen until you are satisfied that you have made all modifications desired.
It has the following submenus:

Induction Variable:
All inductions variables in loop nests in the active unit are substituted by appropriate expressions in the loop variables (if valid) (6.2.2).

Scalar Forward Substitution:
You will be prompted to select a statement which defines a scalar variable. If it is possible, uses of that variable reached by the definition are replaced by the expression on the right hand side of the assignment statement (6.2.1).

Reduction:
The system attempts to recognize standard reductions, e.g. scalar product or vector maximum. These reductions are marked and then replaced in the final code by an appropriate (parallelized) version for the target machine (6.3.7).

Inline Expansion:
You will be asked to select the subprogram call as described in 5.2.3.3. The called subprogram is then inserted textually at this position in the program (6.2.5).

Loop Distribution:
This transformation splits a single DO loop with multiple statements into several separate DO loops. It is, in particular, important for vectorizing DO loops (6.3.1). It is executed for all loop nests in the active unit.

Loop Fusion:
You may combine the statements of two loops at the same level to save overhead under certain conditions. You will be asked to select the DO loops.

Loop Interchange:
You will be asked to select the DO loops to be interchanged one by one. They will be exchanged in the program code if the transformation is valid (6.3.3).

Loop Unrolling:
You will be asked to select the DO loop, and then to input the unroll depth. This is generally used to improve cache performance (6.3.4).

Unroll Jam:
This transformation is applied to a pair of loops at the same level. You will be asked to select them one by one (6.3.5).

Strip Mining:
You will be asked to select a DO loop and then to specify the blocking factor. An additional loop is created in the code – this can improve the locality of your code (6.3.6).

Constant Propagation:
Interprocedural constant propagation is performed (6.2.3).

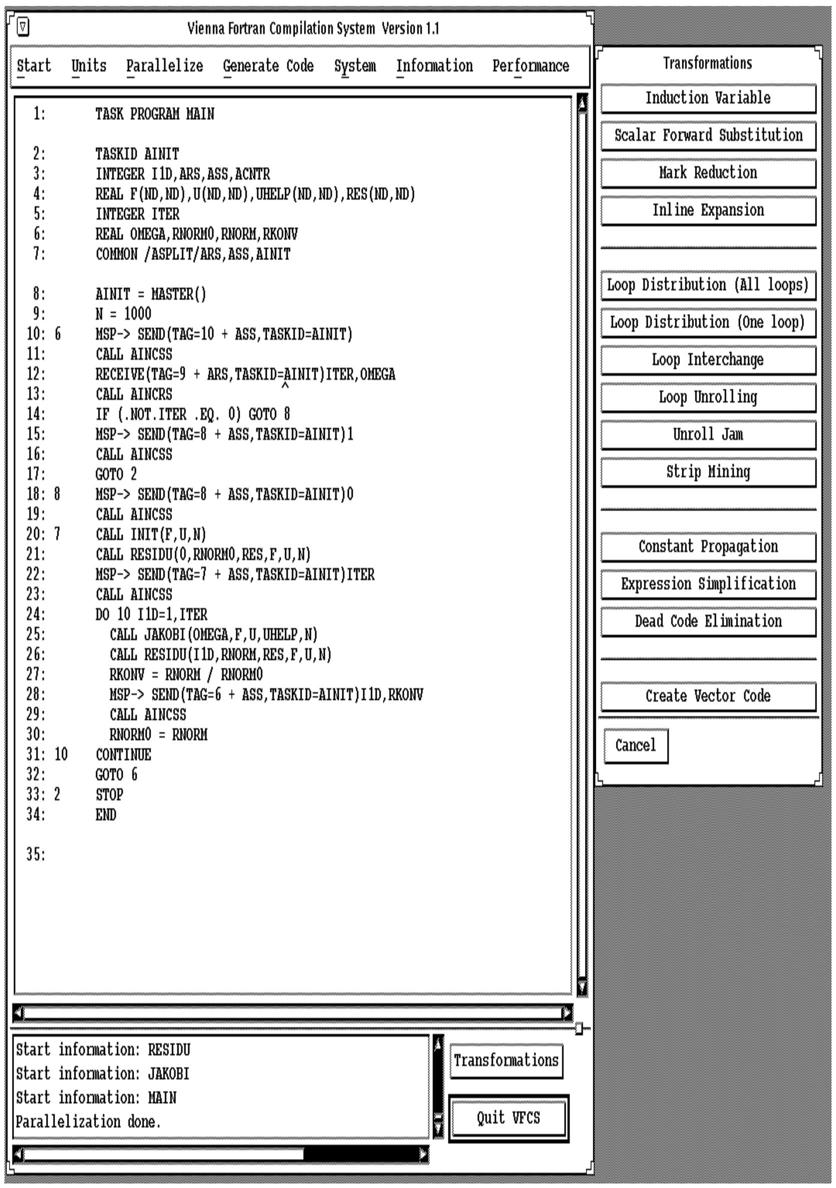


Figure 5.19: Program transformation menu

Expression Simplification:
The expression simplifier is executed for the active unit. Arithmetic expressions are simplified and standardized.

Dead Code Elimination:
Code which has no effect on the outcome of the program is eliminated in the active unit (6.2.4).

Create Vector Code:
Use this to vectorize the DO loops in a program unit where the transformation is valid.

5.4 Preparing Your Program for Parallelization

Before you submit a program to VFCS, you should remove any of the following constructs and check that it adheres to the current system limitations. These limitations will be removed where possible in future versions.

- VFCS currently accepts ANSI Fortran only. It is very strict in this respect. In particular, namelist I/O and Include files are not handled, nor are some other common Fortran extensions permitted. (REAL*8?) Check your program with your Fortran compiler and -ANSI switch to find these constructs before starting VFCS.
- VFCS should not be used to debug your Fortran code. The research system rejects any erroneous program, but in a very unsophisticated manner. Please ensure that your program compiles on a workstation first (although this may not guarantee that the code is syntactically correct!). See a short list of Fortran errors below.
- 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 behaviour of your program. If a memory fault should occur when applying a transformation to a relatively large subroutine, then you should also 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 does not accept an I/O statement 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.
- VFCS does not accept an program if a subroutine passed as an argument to another subroutine itself performs I/O. This is again a restriction imposed by the splitting phase.
- 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.

5.5 Example

The following pages show the steps taken to parallelize a Jacobi program. The screens are assumed to be self-explanatory.

Chapter 6

Transformations and Information Services

In this chapter, we describe the *transformation catalog* and the *information services* of the Vienna Fortran Compilation System. 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).* We will refer to the entirety of information sets computed for a program (in particular, data flow and dependence information) as the *program information set.*

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.

6.1 Frontend Transformations

The frontend performs traditional compiler analysis (scanning, parsing and semantical analysis) as well as program normalization (cf. Fig. 6.1). The normalization transformations include:

- Substitution of statement functions
- Transformation of EQUIVALENCE statements so that only scalar variables or only array elements share the same storage. Different EQUIVALENCE statements are combined.
- Moving all FORMAT and DATA statements to the end of the declaration part of the program unit
- Substitution of COMPUTED GO TO , ASSIGNED GO TO and ARITHMETIC IF statements using conditional GO TO statements
- Substitution of BLOCK IF statements by LOGICAL IF statements
- Substitution of RETURN statements (inside do-loops) by conditional GO TO statements
- Termination of DO statements by CONTINUE statements

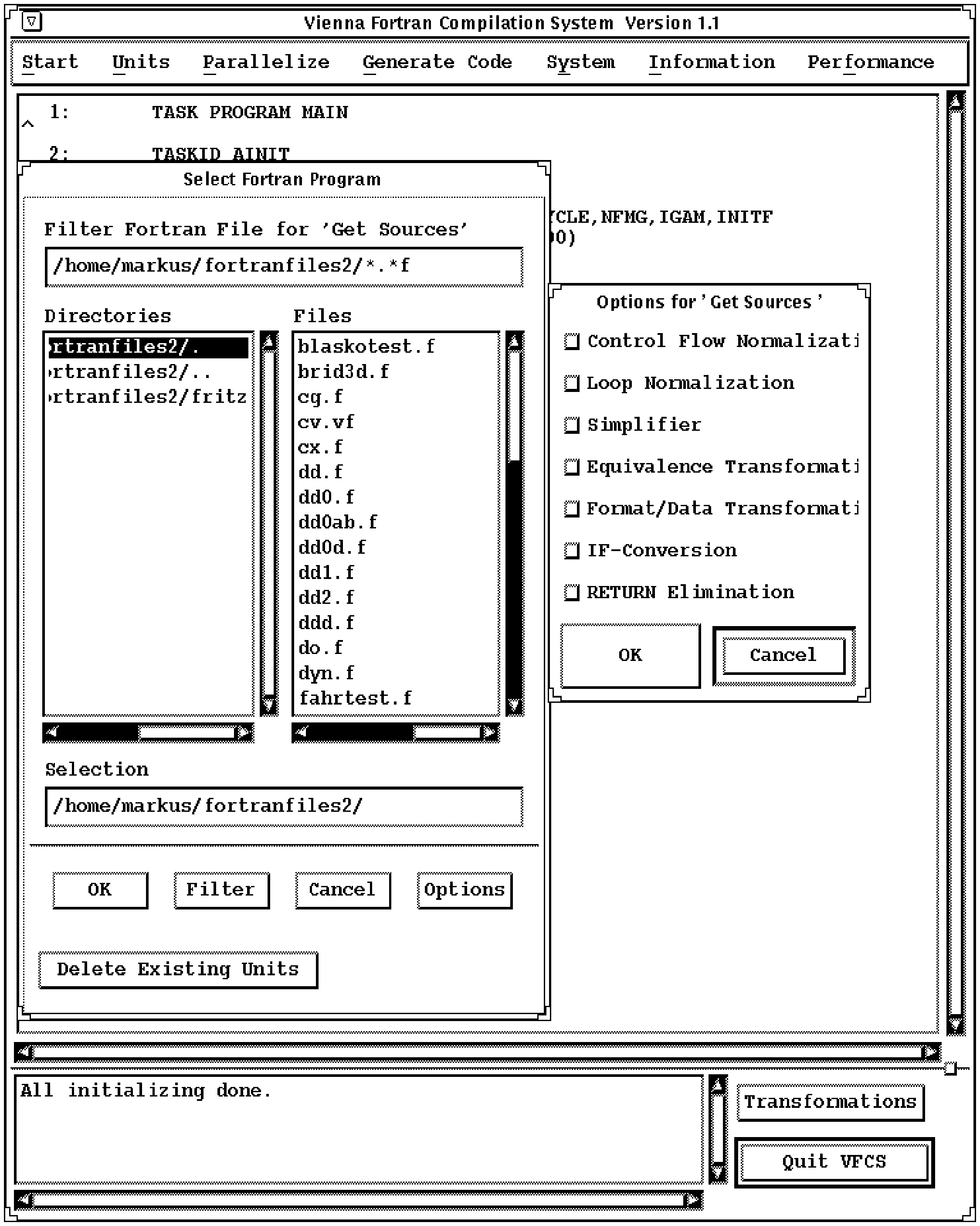


Figure 6.1: Frontend transformations

- Normalization of do-loops such that the step-size becomes 1
- If conversion
- Standardization of subscript expressions
- Elimination of unreachable code
- Elimination of unreachable labels
- Elimination of useless CONTINUE statements

In the following, we illustrate some of these transformations by examples.

Example: In-line Expansion of Statement Functions

```
      program stmtfun
      real x,y,a(10)

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

      read(*,*) y

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

      write(*,*) a

      end
```

```
is transformed into:

      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
10    CONTINUE
      I = I1D
      WRITE(*,*) A
      END
```

Example: The EQUIVALENCE Transformation. This transformation presupposes interprocedural constant propagation.

```
      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
10    continue

      ....
```

is transformed into:

```
      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
10    CONTINUE

      .....
```

Example: Placement of FORMAT and DATA Statements.

```
      program datast

      integer a(10)

      read(*,*) x

      data A/10*1/

      do 10 i=1,10
         a(i) = a(i) * X
10    continue
```

```
      write(*,33) x

33    FORMAT(F10.3)

      end
```

is transformed into:

```
      PROGRAM MAIN

      INTEGER I1D
      INTEGER A(10)

33    FORMAT(F10.3)     <===
      DATA A/10 * 1/    <===

      READ (*,*)X
      DO 10 I1D=1,10
         A(I1D) = A(I1D) * X
10    CONTINUE
      I = I1D
      WRITE(*,33) X
      END
```

Example: Transformation of Computed GOTO

```
      program compgoto

      integer a,b

      read(*,*) a

      goto (10,20,30) a

10    write(*,*) '10'
20    write(*,*) '20'
30    write(*,*) '30'

      end
```

is transformed into:

```
      PROGRAM MAIN

      INTEGER CG1
      INTEGER A,B
```

```
      READ (*,*)A
      CG1 = A
      IF (CG1 .EQ. 1) GOTO 10
      IF (CG1 .EQ. 2) GOTO 20
      IF (CG1 .EQ. 3) GOTO 30
10    WRITE(*,*) '10'
20    WRITE(*,*) '20'
30    WRITE(*,*) '30'
      END
```

Example: Transformation of Arithmetic GOTO

```
       program arif

       read(*,*) i

       if (i) 10,20,30

10     write(*,*) '10'
20     write(*,*) '20'
30     write(*,*) '30'

       end
```

```
is transformed into:
```

```
      PROGRAM MAIN

      DOUBLE PRECISION AIF1

      READ (*,*)I
      AIF1 = I
      IF (AIF1 .LT. 0) GOTO 10
      IF (AIF1 .EQ. 0) GOTO 20
      IF (AIF1 .GT. 0) GOTO 30
10    WRITE(*,*) '10'
20    WRITE(*,*) '20'
30    WRITE(*,*) '30'
      END
```

Example: Transformation of BLOCK IF into LOGICAL IF

```
      IF (I.GT.0) THEN
```

```
         CALL SUB1(A)
      ELSE
         CALL SUB2(A)
      ENDIF
      WRITE(*,*) A,B,C
```

is transformed into:

```
      IF (.NOT.I .GT. 0) GOTO 3
      CALL SUB1(A)
      GOTO 2
3     CALL SUB2(A)
2     WRITE(*,*) A,B,C
```

Example: Transformation of RETURN statements. RETURN statements occurring at the end of procedures will be eliminated; RETURN statements inside a do loop will be substituted by a conditional GOTO statement.

```
      SUBROUTINE SUB(X)
      REAL X(100)

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

      RETURN
      END
```

is transformed into:

```
      SUBROUTINE SUB2(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
10    CONTINUE
      IF (.NOT.EX1) GOTO 1
      ...
1     END
```

Example: DO Loop Termination. DO loops are transformed in such a way that each DO loop is terminated by a loop-specific CONTINUE statement.

```
      DO 100 I=1,100
         DO 100 J=1,100
100         A(I,J) = V
```

```
is transformed into:
```

```
      DO 101 I1D=1,100
         DO 102 J1D=1,100
            A(I1D,J2D) = V
102      CONTINUE
101   CONTINUE
```

Example: DO Loop Normalization. DO loops are normalized by introducing a new loop variable and enforcing a lower bound of 1 and step size 1.

```
      DO 10 I=2,100,2
         DO 20 J=1,10
            C(I,J) = V
20       CONTINUE
         DO 30 J=1,20
            D(I,J) = V
30       CONTINUE
10    CONTINUE
```

```
is transformed into:
```

```
      DO 10 I1D=1,50
        DO 20 J2D=1,10
          C(2 * I1D,J2D) = V
20      CONTINUE
        DO 30 J2D00=1,20
          D(2 * I1D,J2D00) = V
30      CONTINUE
10    CONTINUE
```

Example: Standardization of Subscript Expressions. 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.

```
      DO 10 I=1,10
         DO 20 J=1,10
            DO 30 K=1,10
               A(I) = A(4*K + 3*J + 2*I + I -99 + J)
30          CONTINUE
20       CONTINUE
10    CONTINUE
```

is transformed into:

```
      DO 10 I1D=1,10
        DO 20 J2D=1,10
          DO 30 K3D=1,10
            A(I1D) = A((-99) + 3 * I1D + 4 * J2D + 4 * K3D)
30        CONTINUE
20      CONTINUE
10    CONTINUE
```

Example: Handling of Expressions With Side Effects

```
      Y = X + F(S)
```

where the reference to F(S) results in an assignment side effect, is transformed into:

```
      FR = F(S)
      Y = X + FR
```

Example: Logical Expression Simplification

```
     IF (.TRUE.) CALL SUB()
```

is transformed into:

```
     CALL SUB()
```

6.2 Standard Transformations

Standard transformations (see Fig. 6.2) have the purpose of adapting the program in such a way that loop-based transformations (Section 6.3) applied subsequently yield better results. They include:

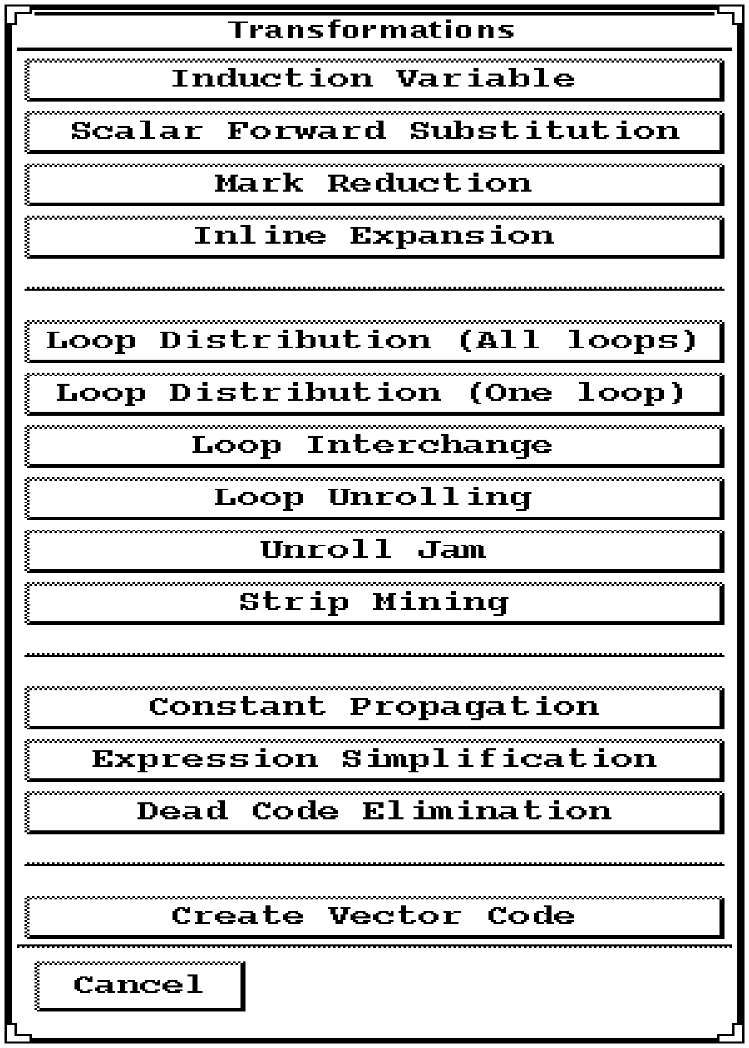


Figure 6.2: Transformation menu

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

Below, we briefly characterize the applicability and the effect of these transformations, and illustrate them by a number of examples.

6.2.1 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: Scalar Forward Substitution

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

is transformed into

```
      DO 10 I=1,10
         A(2*I+1)=A(2*I)+5
10    CONTINUE
```

6.2.2 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: Induction Variable Substitution

```
      DO 10 I=1,10
         K=1
         DO 20 J=1,10
            K=K+2
            B(I,J)=B(I,J+K)+1
20       CONTINUE
10    CONTINUE
```

is transformed into

```
      DO 10 I=1,10
         DO 20 J=1,10
            B(I,J)=B(I,1+3*J)+1
20       CONTINUE
10    CONTINUE
```

6.2.3 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: Constant Propagation

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

is transformed into

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

6.2.4 Dead Code Elimination

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

Example: Dead Code Elimination

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

is transformed into

```
      DO 10 I=1,10
         DO 20 J=1,10
            B(I,J)=B(I,11)+1
20       CONTINUE
10    CONTINUE
```

6.2.5 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: Inline Expansion

```
      PROGRAM MAIN

      REAL A(1000),C(1000),E(1000)

      DO 50 L=1,100
         A(2*L-1)=4*C(L)
         CALL ISDF( A(-1+2*L), E(L), C(L) )
50    CONTINUE
      END

      SUBROUTINE ISDF( X, Y, Z )

      REAL X(2),Y,Z

      Y=X(1)**3-5*X(2)
      Z=3*X(1)**2-5
      RETURN
      END
```

is transformed into

```
      PROGRAM MAIN

      REAL A(1000),C(1000),E(1000)
      INTEGER ARGEXP,ARGE00,ARGE01,OFFDIM

      DO 50 L=1,100
        A(-1+2*L)=4*C(L)
        ARGEXP=-1+2*L
        ARGE00=L
        ARGE01=L
        OFFDIM=ARGEXP-1
        E(ARGE00)=A(1+OFFDIM)**3-5*A(2+OFFDIM)
        C(ARGE01)=3*A(1+OFFDIM)**2-5
50    CONTINUE
      END
```

6.3 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.

6.3.1 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: Loop Distribution

```
      DO 10 I=1,10
         C(2*I)=5
         DO 20 J=1,10
            B(I,J)=C(2*I)+A(I,J-1)
            A(I,J)=B(I-1,J)+1
20    CONTINUE
10
   CONTINUE
```

is transformed into

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

This can be further transformed into vector code:

```
      C(2:20:2)=5
      DO 12 I=1,10
         A(I,1:10)=B(I-1,1:10)+1
         B(I,1:10)=C(2*I)+A(I,0:9)
12    CONTINUE
```

6.3.2 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: Loop Fusion

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

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

is transformed into

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

6.3.3 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: Loop Interchange

```
      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
```

is transformed into

```
      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
```

6.3.4 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: Loop Unrolling

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

can be transformed by unrolling with depth 2 into

```
      DO 10 I=1,999,2
               A(I)=B(I+2)*C(I-1)
               A(I+1)=B(I+3)*C(I)
10    CONTINUE
```

6.3.5 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: Unroll and Jam

```
      DO 11 J1D=1,200
        DO 12 I2D=1,1000
          A(I2D,J1D) = A(I2D,1 + J1D) + A(I2D,(-1) + J1D)
12      CONTINUE
11    CONTINUE
```

can be transformed into

```
      J00 = MOD(200,2)
      DO 11 I00=0,99
        J1D = I00 * 2 + 1 + J00
        DO 12 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)
 12       CONTINUE
 11     CONTINUE
```

6.3.6 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: Strip Mining

```
        DO 11 J1D=1,200
          DO 12 I2D=1,1000
            A(I2D,J1D) = A(I2D,1 + J1D) + A(I2D,(-1) + J1D)
 12       CONTINUE
 11     CONTINUE
```

can be transformed by strip mining with block length 10 to

```
        DO 11 J1D=1,200
          DO 12 I00=0,100
            I01 = I00 * 10 + 1
            DO 13 I2D=I01,MIN(1000,10 - 1 + I01)
              A(I2D,J1D) = A(I2D,1 + J1D) + A(I2D,(-1) + J1D)
 13         CONTINUE
 12       CONTINUE
 11     CONTINUE
```

6.3.7 Reduction

Recurrence 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: Reduction

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

is transformed into

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

6.4 Parallelization Transformations

The first two subsections deal with the transformations supporting Phase 3 (initial parallelization) and Phase 4 (optimization and target code generation) of the basic parallelization strategy, as outlined in Chapter 4. Finally, subsection 6.6 discusses the FORALL loop and its use.

6.4.1 Initial Parallelization

At the beginning of this phase, a program has been analyzed, normalized, and split into a host and a node program. Furthermore, the data distribution for that program has been completely specified (either as a part of a Vienna Fortran program, or interactively, by using the data distribution language). Initial parallelization is performed by applying the following three steps to the node program: *Distribution analysis*, *masking* and *communication insertion*. It results in the creation of the *defining parallel program*.

6.4.1.1 Distribution Analysis

In VFCS 1.0, the distribution of all arrays which are not dummy arguments is determined explicitly by the user. A dummy argument inherits the distribution of the corresponding actual argument. *Distribution analysis* determines the set of all distributions that can be bound to a dummy argument at execution time, and performs transformations which simplify the procedure interface, if necessary. Each distribution is internally associated with a *distribution number*.

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 asociated with that particular incarnation. If a dummy argument can be bound in different call vectors to a replicated as well as a non-replicated actual argument, this conflict is solved by **cloning**: two different versions of the procedure are generated. This is done automatically.

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. 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.

In the following, we discuss in more detail the classification of dummy argument arrays. Such arrays are classified to be either *standard non-replicated arrays* having standard distributions in all incarnations and *non-standard non-replicated arrays* which may have non-standard distributions in some incarnations.
To determine this property, all calls to the procedure are examined. For a given incarnation, if the actual array is a standard non-replicated 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.
Example:

```
        REAL A(100,100,100)

                ...
        DO 10 I=1,100
           CALL F(A(1,1,i))
10      CONTINUE


        SUBROUTINE F(B)
        REAL B(100,100)

                ...
        END
```

The dummy array B is a subarray of the actual array A in all incarnations of F.

If the dummy array is adjustable, then the above property cannot always be determined automatically. In such a situation the user is prompted for that information.

The system presents the MIMD Information window (cf. Fig. 6.3) on the screen. This window can be handled in the usual way. The dummy array, the actual array, the CALL statement and the declaration of the dummy array are presented. The user may now select one of the following buttons:

Yes	Yes, it is always a subarray
No	No, it may not be a subarray in at least one incarnation
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.
Exit	Cancel the transformation.

Computing Call Vectors With Virtual Arrays
Distribution analysis can be performed automatically in the absence of work space arrays (as describe in 3.2.3). 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.

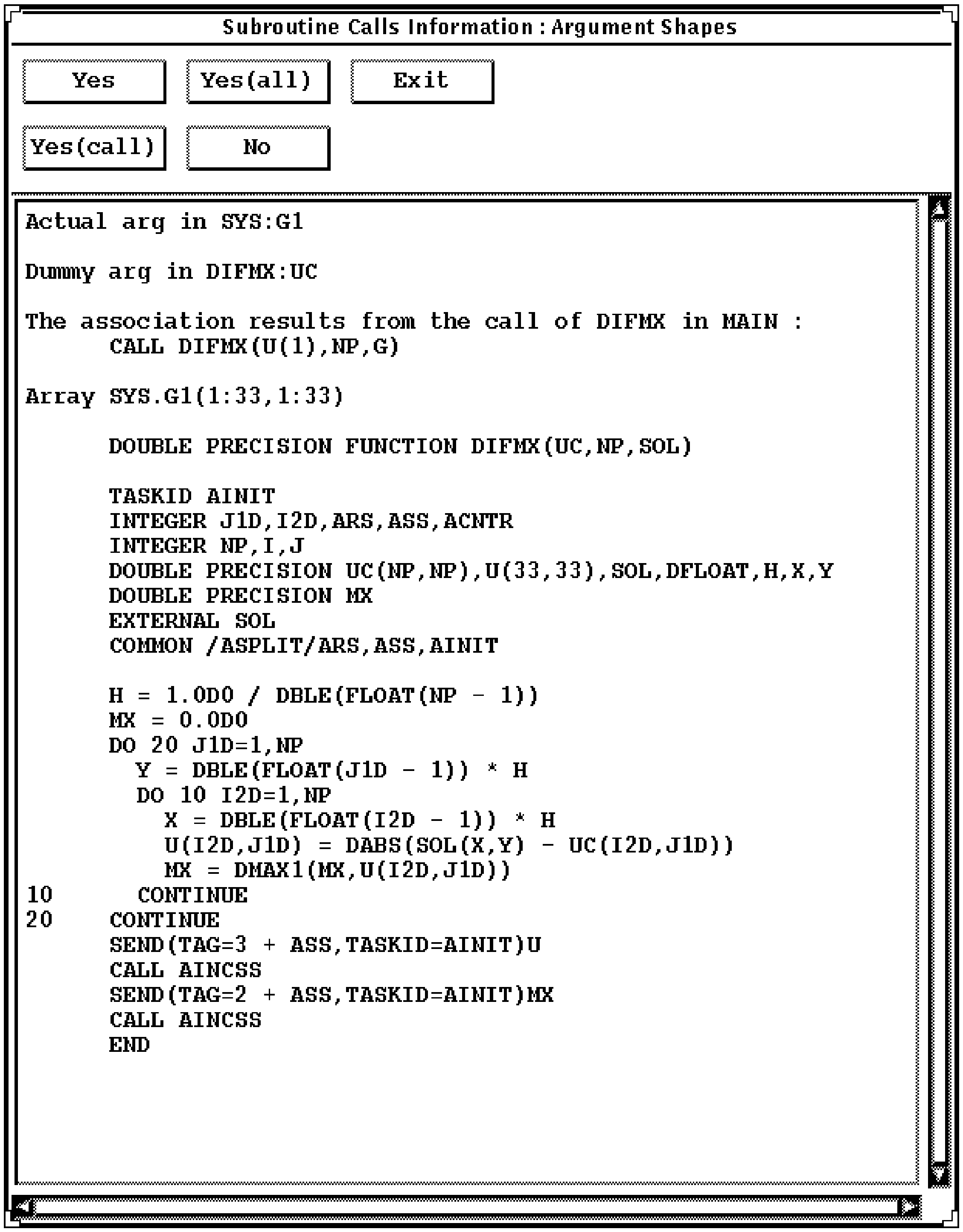


Figure 6.3: The MIMD information window

Example:
Let U be a workspace with the virtual arrays G1 and G2 in the manner described in 3.2.3. The following code implements the restriction operation in a multigrid computation. The subroutine computes new values for the grid points of the next coarser grid from the grid points of the fine grid. The fine grid is implemented by GF and the coarser grid by GC.

```
      REAL    U(40000)

      DO 20 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,...)
         ...
 20   CONTINUE


      SUBROUTINE RESTR (...,GF,NPF,GC,NPC...)
      REAL GF(NPF,NPF),GC(NPC,NPC)

      ...
      END
```

When analyzing the call statement, the system is unable to determine which virtual arrays are accessed via the dummy parameters GF and GC.

If the VFCS detects such a situation during the computation of the call vectors, it requires the user's assistance. The systems presents the user with a window containing (cf. Fig. 6.4):

- the call statement
- the work spaces and their virtual arrays being accessed
- 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.

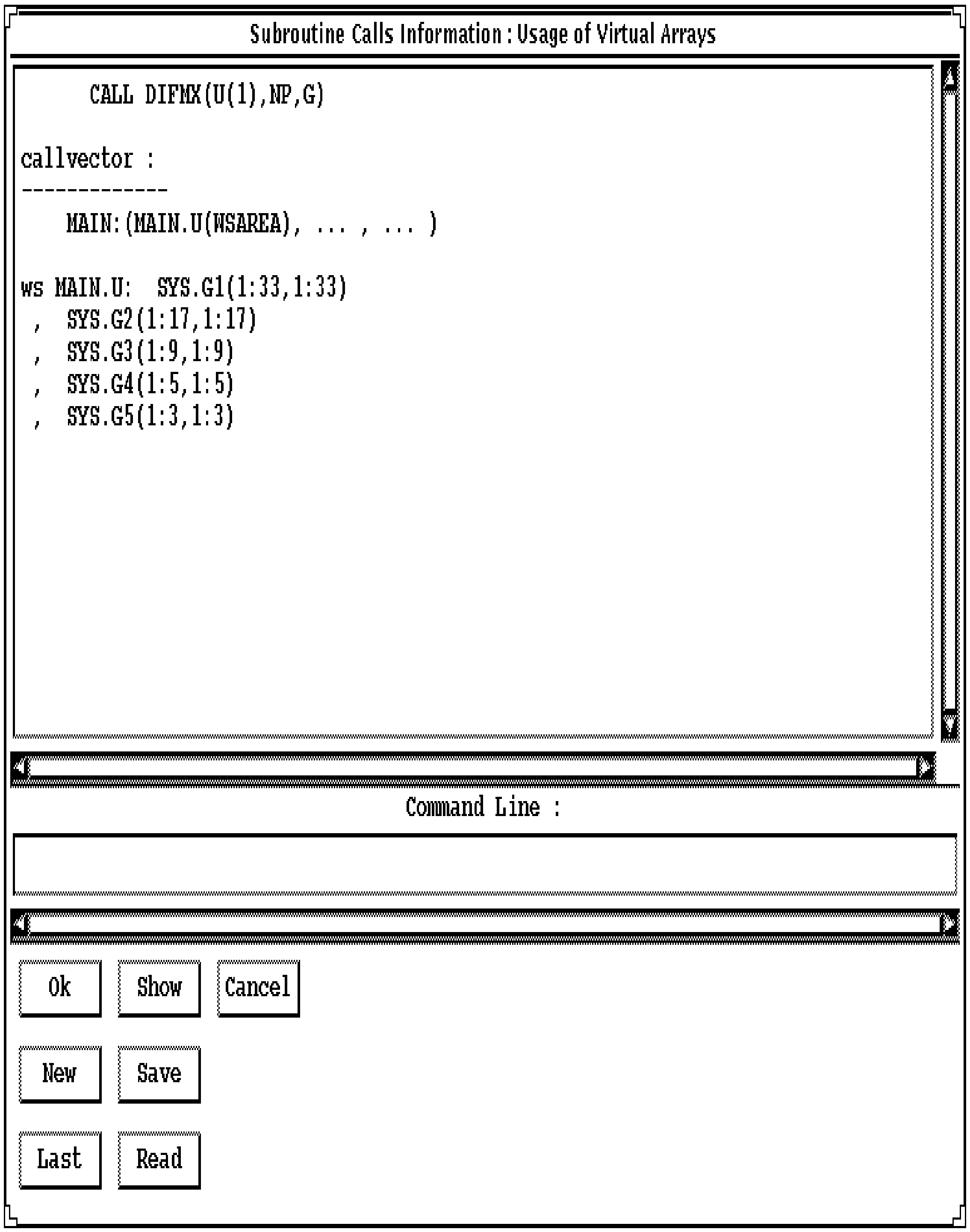


Figure 6.4: Subroutine calls and virtual arrays

Save	Starts the protocol for all supplied combinations. The system requires a file name.
Read	If the information needed to compute the call vectors 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.

Example:
The user may specify the combinations for the previous example by:

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

Sequences are 1–2 and 2–3.

Both specifications determine the same combinations

```
(G1     G2)
(G2     G3)
```

6.4.1.2 Masking

For each statement except FORALL a *mask* is determined which suppresses the execution of the statement if an array element of a non-replicated array is written which is not local to the executing process (see Section 4). Since the subsequent Phase 4 will optimize the defining parallel program, the source code of the units is not actually modified by Phase 3: the masks are stored in the intermediate representation.

Mask propagation is a transformation that eliminates the execution of assignments to scalar variables if the computed values are not needed by the executing process. It is selected from **[Parallelize : Mask Propagation]**.
Example:

```
        REAL A(100)

        DO 10 I=2,99
            S=A(I-1)+A(I+1)
            A(I)=2*S
10      CONTINUE
```

If A is a non-replicated array, the assignment to A is only executed if a local array element is written. In contrast, the assignment to the scalar variable S is executed in each node process. The value of S is needed only in those iterations which determine a new value for a local array element of A. Thus the mask of the second statement can be propagated to the first statement:

```
        DO 10 I=2,99
           S=A(I-1)+A(I+1)      mask: m2
           A(I)=2*S             mask: m2
10      CONTINUE
```

Mask propagation is only performed within a given DO-loop.

6.4.1.3 Communication Insertion

All references to non-replicated arrays in a program are treated as potentially non-local references, as described in Section 4, and EXCH statements that organize the communication are determined. As before, the source code of the units is not actually modified.

6.4.2 Optimizing Transformations

Optimizing transformations fall into two categories:

- Optimization of communication
- Optimization of masking

6.4.2.1 Optimizing Communication

This transformation extracts communication from DO-loops and thus combines single element messages to larger ones. It is selected by **[Parallelize : Optimize Communication]**.
Example:

```
        REAL A(100,100)

        DO 20 I=1,99
           DO 10 J=1,100
S1:             A(I,J)= ...A(I+1,J)     com: A(I+1,J) {0/1,0/0}
10         CONTINUE
20      CONTINUE
```

In the defining parallel program, the communication to update the copies of the boundary elements of the right neighbor is inserted directly before $S1$ and thus the copies are updated by single element messages. The optimization transformation extracts the communication from surrounding DO-loops and implements the updating of all copies in a node process via a single large message. The communication is implemented in the intermediate representation.

Example:

```
        REAL A(100,100)

        DO 20 I=1,99                com:  A(*,*) {0/1,0/0}
           DO 10 J=1,100
               A(I,J)= ...A(I+1,J)
10         CONTINUE
20      CONTINUE
```

6.4.2.2 Optimizing Masking

This transformation enforces the masks of statements in a DO-loop – under certain conditions – by an appropriate adaptation of the loop bounds. It is selected in [**Parallelize : Optimize Masking**].

Example:

```
        REAL A(100,100)

        DO 20 I=1,99
           DO 10 J=1,100
S1:             A(I,J)= ...         mask: m1
10         CONTINUE
20      CONTINUE
```

In the defining parallel program, the execution strategy is enforced by the statement masks. Each process has to execute the entire loop and statement $S1$ is only executed in an iteration if the array element written is local to the executing process. To suppress the execution of $S1$ in other iterations, these iterations can be eliminated by an adaptation of the loop bounds. Thus the optimizing transformation determines restrictions for the loop variables and eliminates the mask of $S1$.

Example:

```
        REAL A(100,100)

        DO 20 I=1,99                loop restr:    $L1 <= I <= $R1
           DO 10 J=1,100            loop restr:    $L2 <= J <= $R2
               A(I,J)= ...          no mask
10         CONTINUE
20      CONTINUE
```

The local segment of the executing node process is A($\$L1 : \$R1, \$L2 : \$R2$).

6.5 Information Services

The *information services* of VFCS 1.0 compute analysis information and provide access to the program information set. In this section, we outline the major categories of information sets and the methods for access. Information services included in VFCS are:

- data flow analysis
- dependence analysis
- overlap analysis

6.5.1 Dataflow Analysis

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 the menu **[Information : Dataflow]**. 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)

6.5.2 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 the menu [**Information : Dependences**]. This will result in both the computation of the information mentioned above (if the data flow chains do not exist at that time, the dependences for the scalar variables within DO-loops are computed by calling the data flow analysis service (6.5.1) and the creation of an output window (cf. Fig. 6.5). Then the user has two possibilities for displaying dependences:

1. the user can look at all the dependences one statement is involved in by selecting the corresponding line of code;
2. the user can look at all the dependences between all the statements in the body of a DO-loop by selecting the DO-loop.

The dependence type can be selected first:

1. **True dependence**
   This will result in selecting only the true dependences for display for all variables defined in the selected region.
2. **Anti dependence**
   This will result in selecting only the anti dependences for display for all variables used in the selected region.
3. **Output dependence**
   This will result in selecting only the output dependences for display for all variables defined in the selected region.
4. **All dependences**
   This will result in selecting all dependences for display for all variables occuring in the selected region.

Having selected the dependence type, the user can further restrict the dependences to be displayed by selecting a dependence level. This can be done by specifying

1. an integer $i, 1 \leq i \leq 9$

   This will result in displaying only loop carried dependences with level i.
2. the integer 0

   This will result in displaying only loop independent dependences.
3. All

   This will result in displaying all dependences of the selected type.

Every dependence displayed has the form:

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

where

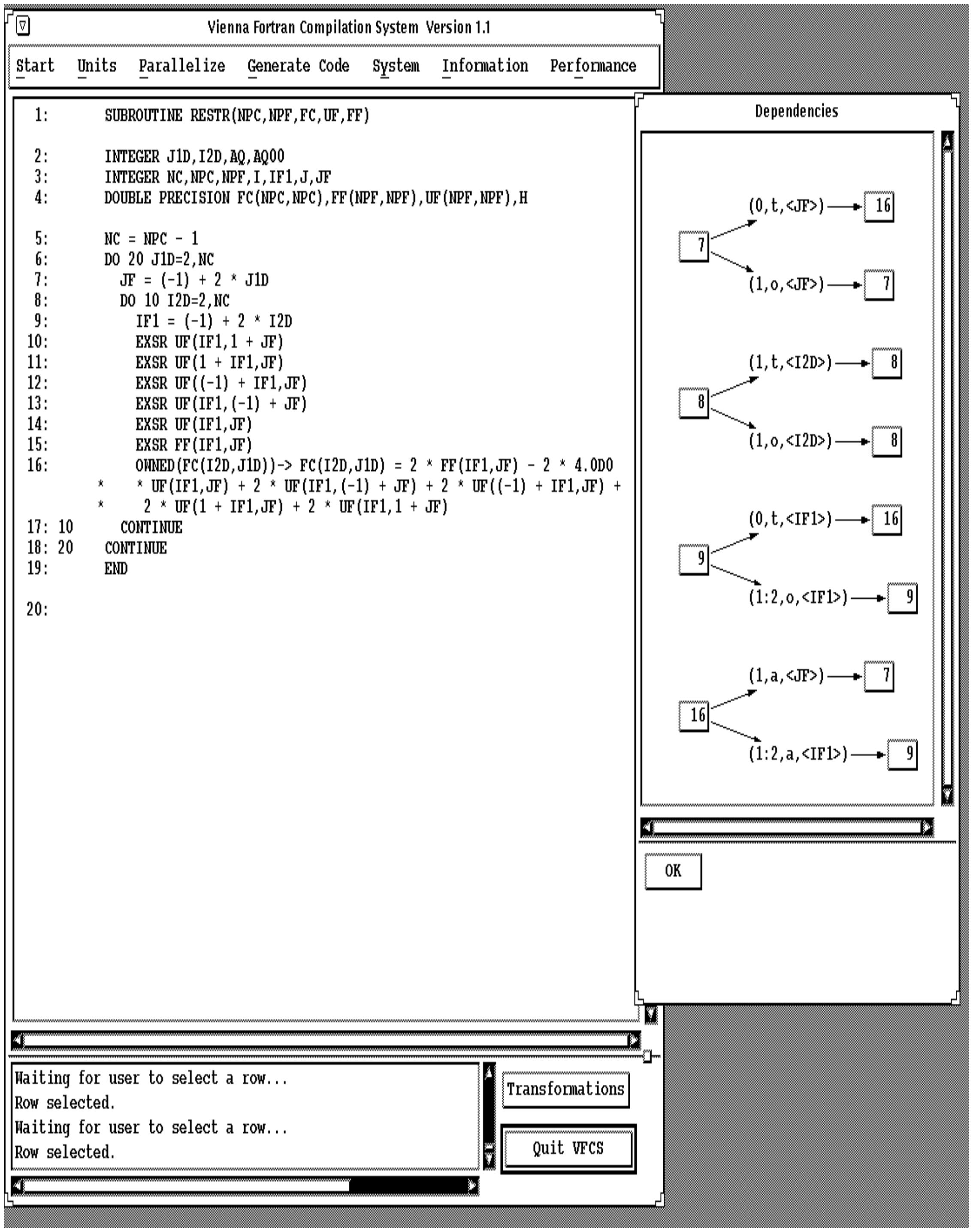


Figure 6.5: Data dependence information

- 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 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

6.5.3 Overlap Analysis and Display

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). The associated system services can be used by selecting the menu [**Information : Overlap Unit**] or [**Information : Overlap All Units**].

Overlap analysis is performed for each reference to a non-replicated array in the actual unit. 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.

There are separate menu items for the computation and display of overlap information. The computed overlap information can be inspected via the [**Information : Show Overlap**] menu. You will not see the desired information via this menu if you have not previously selected the corresponding overlap analysis menu item(s).

6.5.4 Syntax tree

Selecting the menu item [**Information : Syntax Tree**] will display the syntax tree beginning with the statement the cursor is positioned at. Only the pointers dwn1, dwn2 and rght of each node are shown (if not equal NULL). The values of leaf nodes are presented.

6.5.5 Control Flowgraph

Selecting the menu-item [**Information : Flowgraph**] 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.

6.5.6 Symbol Table

Selecting the menu item [**Information : Symboltable**] will display some information, extracted from the symboltable, about the variables accessed.

6.5.7 Global Variables

Selecting the menu item [**Information : Global Variables**] will display the dummy arguments and the variables of COMMON blocks accessed. Every entry is marked as 'use_def' , 'may_def' or 'must_def', dependent on the first access.

6.5.8 I/O lists

Selecting the menu-item [**Information : Use/Def Sets**] 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).

6.5.9 Coefficient Lists

Selecting the menu-item [**Information : Coeff**] will display for every array within a statement (identified by row number and statement type) the following information:

- the name of the array (positioned after the keyword 'ARRAY')
- whether or not a dimension satisfies the linearity condition: there are eight entries after the keyword 'LIN :' which are ordered from right to left, i.e. the rightmost entry corresponds to the linearized index expression (if it exists), the next entry corresponds to the index expression of the first dimension, the next entry corresponds to the index expression of the second dimension, and so on. Each entry is either a 'X' representing the linearity of the corresponding index expression or a '.' representing non- linearity.
- whether or not all of the multiplicative coefficients of a dimension are constant:

  There are eight entries after the keyword 'CONST :' which are ordered from right to left, i.e. the rightmost entry corresponds to the linearized index expression (if it exists), the next entry corresponds to the index expression of the first dimension, the next entry corresponds to the index expression of the second dimension, and so on. Each entry is either a 'X' representing the case that all multiplicative coefficients of the corresponding index expression are constant, or a '.', representing the case that there is at least one symbolic coefficient.

  Furthermore, for every coefficient there is a line of the form:

  ELEMENT dim_nr, coeff_nr : VALUE=const AT_REF=at_ptr
  where

  – dim_nr indicates that the values 'coeff_nr' , 'const' and 'at_ptr' belong to the index expression in dimension 'dim_nr', where $0 \leq dim\_nr \leq 7$ and 0 corresponds to the linearized index expression

– coeff_nr indicates that the values 'const' and 'at_ptr' belong either to the constant coefficient (of the index expression of dimension 'dim_nr'), i.e. coeff_nr has value '0' , or to the multiplicative coefficient (of the index expression of dimension 'dim_nr') corresponding to the 'coeff_nr'-th do-loop surrounding the statement, i.e. $1 \leq coeff\_nr \leq$ maximal depth of a do-loop nest.

– const gives the value of the coefficient (determined by 'dim_nr' and 'coeff_nr') if it is not symbolic; otherwise it has value '0' and at_ptr has a value not equal '0'.

– at_ptr gives the value of the pointer to the syntax tree expression for the coefficient (determined by 'dim_nr' and 'coeff_nr') if it is symbolic;otherwise it has value '0' .

6.5.10 Call Graph

Selecting the menu item [**Information : Callgraph**] will display the call graphs of the host program and the node program (see Fig. 6.6).

6.5.11 Communication

The communication associated with a statement can be visualized (see Fig. 6.7) via the communication entry in the Parallelize menu [**Parallelize :Communication**]. After code generation, the communication is implemented by procedure calls which are inserted before the statement.

Two types of communication are distinguished: single element communication and block communication. Syntactically, the distinction is apparent through the use of array element references in the first, and array sections in the second case. For example:

```
A(i,j+1)        A(*,j)
```

The actual copies of non-local variables which are updated are determined by such references and the overlap area of the accessed array. For all possible pairs of distributions for the array in the mask and the array read, an individual overlap area is determined.
The information window is made up of different types of lines:

- Lines starting with

  ```
  ovp number ==>
  ```

  and

- simple lines.

The lines starting with an ovp number can be further distinguished to be:

- statement lines
  The ovp number is followed by:
  – a statement number in reverse video
  – the array in the mask, which is used to compute the overlap area, or **true**.
  – a reference to a non-replicated array, which determines the type of communication: Block type or single element type.

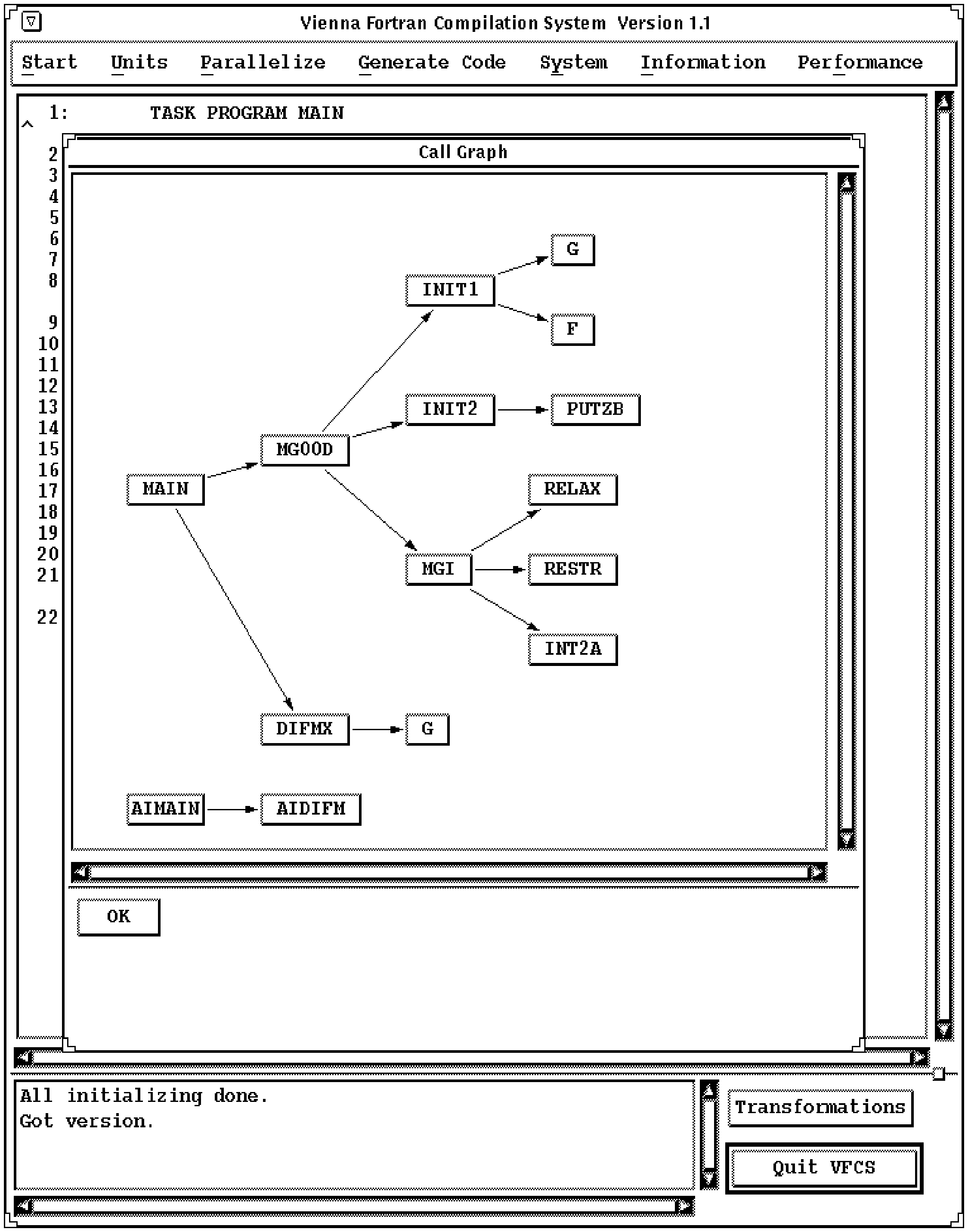


Figure 6.6: Call graph

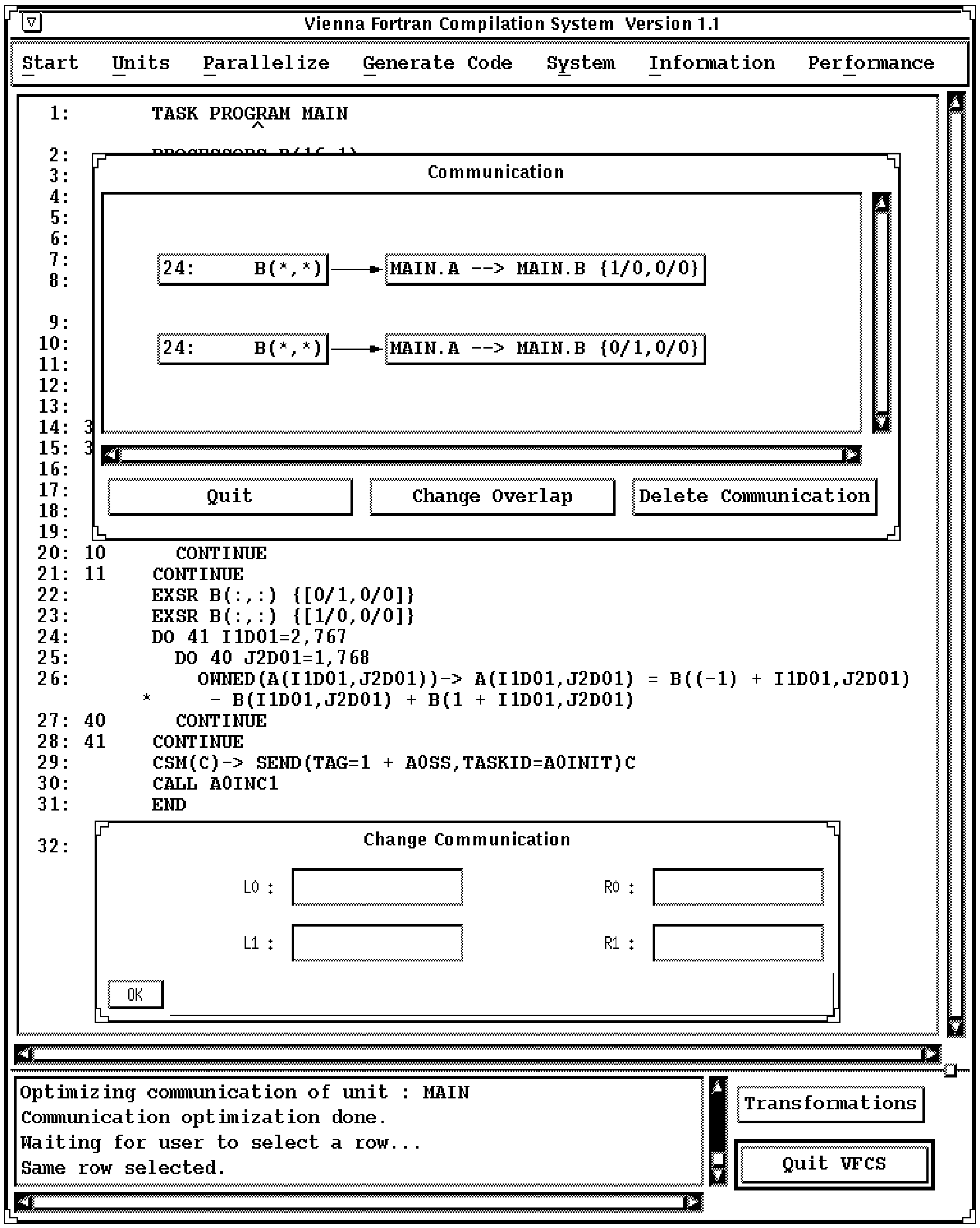


Figure 6.7: Modify communication interactivity

- overlap lines
  Each overlap line following a statement line determines the overlap area for an individual pair of distributions of the array in the mask and the array read. To present more readable information, the names of arrays bound to dummy array parameters are presented instead of the distribution numbers.

Different pairs of non-replicated arrays may determine the same distribution number pair. These additional array pairs are presented in the simple lines following an overlap line.

Example:

```
statement line:  1 ==> 23    B    A(i,j+1)


overlap line:    2 ==> G1  -->  G2 {0/0,0/1}

simple lines:          G3  -->  G4 {0/0,0/1}
simple lines:          G5  -->  G6 {0/0,0/1}
```

This information service can also be used to change the computed overlap information manually. The window is presented with two choices:

- Change Overlap
- Delete Communication

The change ovp entry is selected to change the overlap information belonging to a single statement line. All overlap descriptions, e.g. 0/0,0/1, are set to a user specified value. If it belongs to an overlap line, only the overlap description of this overlap line and following simple lines are changed.

Besides the possibility to change the overlap information, the user can delete the entire communication. In this case, the user has to supply an overlap number of a statement line. The communication which is described by this statement line is then deleted.

6.5.12 Statement Masks

Statement masks can be inspected via the menu item [**Information : Statement Mask**]. Once you select a statement, the mask of this statement is shown.

The possible masks are:

- **LOC**(i)
  The value of i (which may be a constant or a variable) indicates the distribution number of the defined array. If the overlap computation has been performed, those restrictions on the loop variables are presented which have been used to compute the overlap areas.
- **LOOPRESTR**
  The restrictions used to adapt the loop bounds are shown.
- **true**
- **false**

6.6 FORALL Loop Transformation

VFCS 1.0 supports parallel loops called 'FORALL loops'. This kind of loop allows the user to assert that the different instantiations of the loop body are independent and can be logically executed in parallel. To ensure such independence, FORALL loops are not allowed to have any loop-carried 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 FORALL 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.

FORALL loops are particularly useful in situations where the compiler cannot detect the data independence of loop iterations, for example if subscripted variables are used in subscript positions. FORALL loops whose iterations are not independent are illegal and the results from executing such programs are undefined.

The loop body of a FORALL loop may access data items non-local to the processor on which it is executing. Since the execution of an iteration cannot change data items used by any other iteration, any non-local data items used by an iteration can be gathered before the start of the iteration. That is, any communication required for accessing data items owned by other processors can be performed before the FORALL 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 FORALL loop.

6.6.1 FORALL loop body

The body of a FORALL loop can consist of any legal F77 executable statements. In VFCS 1.0, FORALL loops must satisfy the following requirements:

1. No loop-carried dependences are permitted
2. Nested FORALL loops are not allowed
3. There must be at least one access to a non-replicated array
4. Only one-dimensional non-replicated arrays are permitted
5. No non-replicated variable is allowed in subscript functions of non-replicated arrays
6. No input/output, STOP, or RETURN statements are allowed
7. No function and subroutine calls are allowed
8. Scalar variables have the semantics of *private variables* in the sense that they are not live at the end of the FORALL loop (except those in a REDUCE statement)

6.6.2 Work Distribution

Vienna Fortran allows users to control the assignment of loop iterations to processors. This can be done through the optional 'on clause' specified in the FORALL loop header. The 'on clause' consists of the keyword ON followed by either a processor element name or the intrinsic function OWNER, with an array element as argument. For example:

```
FORALL i=low,high  ON $P(i)
```

Here, $P is the default processor array, and iteration i is assigned to the processor $P(i). This is currently not available interactively.

```
FORALL i=low,high  ON OWNER(a(i))
```

The intrinsic function OWNER returns the processor that owns its argument: iteration i is executed on the processor which owns the array element $a(i)$.

The user can also specify a uniform work distribution:

```
FORALL i=low,high  CYCLIC
```

The iteration set is distributed in a round robin fashion across the processors.

```
FORALL i=low,high  BLOCK
```

A contiguous block of iterations is assigned to each processor.

If no work distribution is specified in [**Start Parallelization**], VFCS tries to determine a work distribution depending on the distributions of arrays accessed in the loop body, i.e. an *ON OWNER* work distribution. It looks for the first occurrence of a non-replicated array on the left hand side of an assignment statement in the forall body which fulfills the following restrictions:

1. The assignment is not guarded by a logical if. Otherwise, the array reference may not be a valid reference in all iterations and thus cannot be used in the on clause for the specification of the mapping of all iterations.
2. The array subscript expressions do not contain any variable defined in the forall. This restriction simplifies our implementation since the work distribution can be evaluated at runtime without having to execute loop iterations before.

If such an array reference is found, it is used in the on clause, otherwise, block work distribution is applied. After([**Parallelize : Start Parallelization**]), the user may modify the work distribution provided by the system to any type mentioned above by using [**Parallelize : Modify Work Distribution**]. There is a further option in that menu: the execution of the FORALL loop may be distributed according to the values of an array (cf. 5.3.4.

6.6.3 Resulting Code for a FORALL Loop

Code generation for FORALL loops is performed in [**Code Generation :** *selected target Machine*]. This process combines compile time parallelization techniques with run-time analysis. Information from dataflow analysis, data dependence analysis, data partitioning 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. Each FORALL loop is evaluated in two phases. In the inspector phase, the description of the communication necessary for the loop is generated; in the executor phase this information is used to actually perform the communication and to execute the loop body for the local iteration set.

The **inspector** includes:

1. Construction of a translation table for each distribution class whose members appear in the FORALL loop body.
2. Construction of local iteration sets.
3. Generation of the schedules for nonlocal accesses used in the gather/scatter exchanger routines, and mapping of non-local references to local addresses.

The **executor** includes:

1. Generation of communication using the gather/scatter exchanger routines.
2. Transformation of the FORALL loop body - all non-replicated data are referenced using the local reference strings.

6.6.4 Reduction Operators

As indicated before, FORALL loops in Vienna Fortran are not allowed to have any loop-carried dependences. One consequence of this restriction is that a non-private scalar variable cannot be modified in a FORALL loop. In order to be able to handle recurrences in a parallel loop, the REDUCE statement has been included in Vienna Fortran.

The form of the REDUCE statement is as follows:

```
 REDUCE (<reduction-op>, <variable>, <expression> [, <order>])
```

It specifies a reduction operator (ADD, MULT, MIN, MAX), a target variable and an expression to be accumulated in the target variable. The effect of the statement is to accumulate the values of the expression in all loop iterations in the target variable.
Note that the final value of the variable being used as the target of the reduction is not available until the end of the FORALL loop, and hence cannot be referenced on the right hand side in another statement in the same loop. However, the same variable can be used as a target of multiple reduction statements within the same loop as long as the same reduction operator is used in all cases.
VFCS 1.0 does not specify an order for the individual operations associated with a REDUCE statement. User-defined functions are not allowed as reduction operators.

Example: Vienna Fortran source code

```
        program demo

        parameter (N=100, M=4, KK=N/M)
        processors PPP(M)
        integer ia(N),ib(N)
        real a(N),b(N),y(N) dist(block)

C       Initialization of arrays ia, ib
C       Initialization of arrays a, b, y
C       ...

        forall 10 i=1,N on PPP((i-1)/KK)
           y(i)= a(i) * y(ia(i)) + b(i) * y(ib(i))
10      continue

C       using results of y
C       ...

        end
```

Intermediate code of node-program after [**Start Parallelization**]:

```
        TASK PROGRAM MAIN

        TASKID INIT
        INTEGER I1D,RS,SS,CNTR
        PARAMETER(N=100,M=4,KK=N / M)

        PROCESSORS PPP(M)
        INTEGER IA(N),IB(N)
        REAL A(N),B(N),Y(N) DIST(BLOCK)
        COMMON /SPLI/RS,SS,INIT

        ...

        FORALL 10 I1D=1,N ON PPP((I1D - 1) / KK)
          DISTR ARRAYS: Y,A,B
          Y(I1D) = A(I1D) * Y(IA(I1D)) + B(I1D) * Y(IB(I1D))
10      CONTINUE

        ...

        END
```

Resulting code of node-program after [**Generate Code**]:

```
        PROGRAM MAIN

        PARAMETER(BC1=2,BC2=0)
        INTEGER INIT
        INTEGER I1D,RS,SS,CNTR,SB(1,8),OB(1,6),EACT,IACT,
     *  EXEC(25),ITES(100),SEGM(4),K,OFFD,DUMM(1),ND0,SD0,GD0(25),
     *  LD0(25),NU0,SU0,GU0(50),LU0(50),NU1,SU1,GU1(25),LU1(25),NU2,
     *  SU2,GU2(25),LU2(25),AHOST,PR(4)
        PARAMETER(N=100,M=4,KK=N / M)

        PROCESSORS PPP(M)
        INTEGER IA(N),IB(N)
        REAL A(25 * BC1 + BC2),B(25 * BC1 + BC2),X(25 * BC1 +
     *  BC2),Y(25 * BC1 + BC2) DIST(BLOCK)
        COMMON /SPLI/RS,SS,INIT
        COMMON /SYS/PR,SB,OB,SE00
        COMMON /RUN0/EACT,EXEC,IACT,ITES

        ...

C  Computation of subscript functions
```

```
C-----------------------------------
      IACT = N - 1 + 1
      DO 101 I1D=1,IACT
        K = I1D + 1 - 1
        ITES(I1D) = (K - 1) / KK
101   CONTINUE

C  Construction of local iteration set
C------------------------------------
      CALL LITS(14,1,0,DUMM)

C  Precomputing of globally indexed references
C--------------------------------------------
      ND0 = 0
      NU0 = 0
      NU1 = 0
      NU2 = 0
      DO 102 K=1,EACT
        I1D = EXEC(K)
        ND0 = ND0 + 1
        GD0(ND0) = I1D
        NU1 = NU1 + 1
        GU1(NU1) = I1D
        NU2 = NU2 + 1
        GU2(NU2) = I1D
        NU0 = NU0 + 1
        GU0(NU0) = IA(I1D)
        NU0 = NU0 + 1
        GU0(NU0) = IB(I1D)
102   CONTINUE

C  Generation of schedules and sets of local references
C-----------------------------------------------------
      K = SB(1,2) - SB(1,1) + 1
      CALL BFLO(1,SD0,GD0,LD0,ND0,OFFD,K,SB(1,1),SB(1,2),SB(1,4),
     *          SB(1,5),0,SEGM)
      CALL BFLO(1,SU0,GU0,LU0,NU0,OFFD,K,SB(1,1),SB(1,2),SB(1,4),
     *          SB(1,5),0,SEGM)
      K = SB(1,2) - SB(1,1) + 1
      CALL BFLO(1,SU1,GU1,LU1,NU1,OFFD,K,SB(1,1),SB(1,2),SB(1,4),
     *          SB(1,5),0,SEGM)
      K = SB(1,2) - SB(1,1) + 1
      CALL BFLO(1,SU2,GU2,LU2,NU2,OFFD,K,SB(1,1),SB(1,2),SB(1,4),
     *          SB(1,5),0,SEGM)

C  Gather of non-local array elements
C-----------------------------------
      CALL FFGA(SU0,Y,SB(1,2) - OB(1,1) + 2,SB(1,1) - OB(1,1) + 1)
      CALL FFGA(SU1,A,SB(1,2) - OB(1,1) + 2,SB(1,1) - OB(1,1) + 1)
      CALL FFGA(SU2,B,SB(1,2) - OB(1,1) + 2,SB(1,1) - OB(1,1) + 1)
```

```
C  Code of the FORALL loop
C-------------------------
        ND0 = 0
        NU0 = 0
        NU1 = 0
        NU2 = 0
        DO 10 K=1,EACT
          Y(LD0(ND0+1)) = A(LU1(NU1+1)) * Y(LU0(NU0+1)) + B(LU2(NU2+1)) *
     *                    Y(LU0(NU0+2))
          ND0 = ND0 + 1
          NU0 = NU0 + 2
          NU1 = NU1 + 1
          NU2 = NU2 + 1
10      CONTINUE

C  Scatter of updated array elements
C----------------------------------
        CALL FFSC(SD0,Y,SB(1,2) - OB(1,1) + 2,SB(1,1) - OB(1,1) + 1,0)

        ...

        END
```

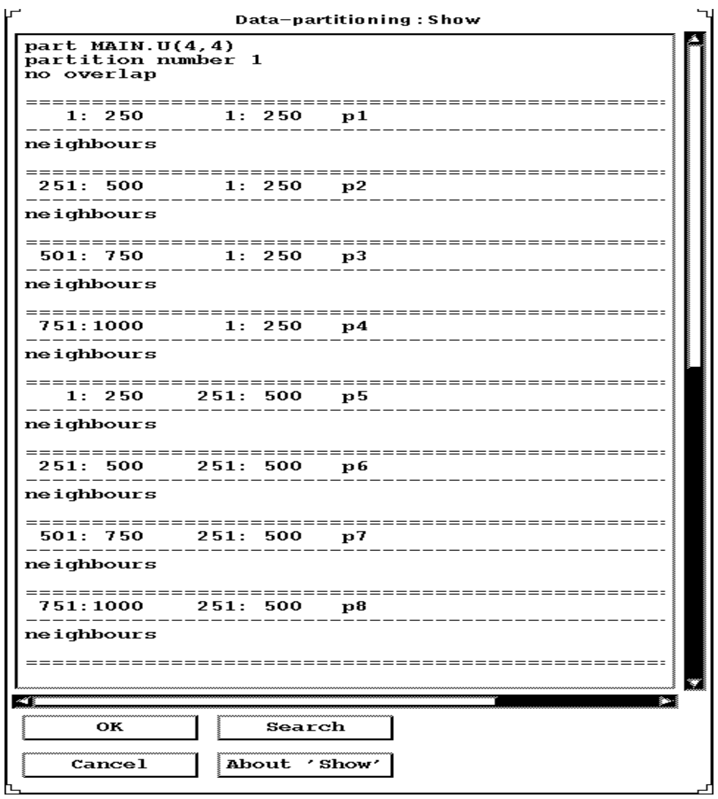


Figure 6.8: Show data partioning information

Chapter 7

The Batch Facility

VFCS 1.0 provides two modes of operation, the *interactive mode* and the *batch mode*. This chapter discusses the use of the batch facilities provided for use within batch mode. It describes the features which are provided, explains how to create and execute batch files both manually and automatically, and how to use the Command Interpreter window.

In batch mode, a file of commands written in the VFCS *batch language* may be automatically applied to a Fortran program. The transformation sequence of a parallelization session may be automatically or manually stored in a text file, modified, and used as a *batch file* in later VFCS runs. Batch files can be created by means of a UNIX text editor, or by protocolling a terminal session (that is, by recording the work done in the interactive mode; see 7.1.2 below). Both use the same language constructs, and a protocol may thus be modified manually. While the interactive mode currently is the dominant paradigm for using VFCS, there are situations where the automatic application of transformation sequences in the batch mode offers significant advantages. This applies in particular if

- minor modifications of an input program do not affect the parallelization strategy
- transformation sequences, encapsulated as *macro sequences*, are applied to more than one program

The **batch facility** of VFCS 1.0 consists of

- a formal language, the batch language, to specify commands for the transformation of programs
- an *interpreter* to execute these commands
- a *system component* to protocol the interactively specified commands.

Most of the commands of VFCS 1.0 can be formulated in the *batch language*. It 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 DO loop)
- commands which are applied to the selected construct.

The batch language is described in the Appendix.

7.1 Creating Batch Files in VFCS 1.0

7.1.1 Manual Creation of Batch Files

In this section, we discuss some constraints on the use of batch commands that must be heeded if you manually create a batch file. The syntax of the batch language is fully described in Appendix B. We use the terminology of that section without further explanation here. Note, however, that the selection of program constructs is performed in the batch language by the positioning of a hypothetical cursor. Hence we refer to this "cursor" in the descriptions below.

- All "transform" commands assume that one of the units is active – in batch terminology, the unit is *loaded* – and produce an error message otherwise. In the case of an error, a subshell is created by the VFCS error handler. The user may return to VFCS (after correcting the batchfile or a similar action) by typing "Control-Z".
- For the "inlinexp" command, the "cursor" must be positioned at the procedure identifier of the CALL statement to be transformed.
- The "loopexc" command for exchanging two loops is expressed by
  - positioning the cursor on the first DO loop
  - marking it by the "mark" command
  - moving the cursor to the second DO loop
  - calling "loopexc".
- The two file names are optional for "start_mimd". They are used to refer to a ".ws" and an ".adj" file, respectively.
- All commands calling front_end_1 (that is, "fe1_new", "fe1_add", and "front_end") must contain an "options" parameter. Observe that this parameter must always begin with "-", even if no explicit options are given.
- A number of the "transform" commands may not be used once the back end, "be", has been called.
- The "copy_unit" command should only be used to create copies for data security. Do NOT use this copy instead of the original program unit during the batch processing. (If you mistakenly use it to test a transformation, you should then use another "copy_unit" command to copy results back to the original unit.
- The vector reductions marked with "mark_reduct" are transformed by invoking the "vcodegen" command.
- The "batch" command is not recursive: if a batch file that is executing is called again, a system error occurs which results in the writing of a core file and return to the UNIX shell.

7.1.2 Automatic Generation of Batch Files

In order to repeat the transformation steps of a VFCS session, the commands can be recorded in a protocol file using the same syntax as a manually constructed batch file. Thus it can be read and modified by means of a text editor. Recording is started by selecting

prot <protocol_file_name>

or

rprot <protocol_file_name>

(see below for further explanation) in the command menu. Only one <protocol_file_name> can be selected; it designates the <protocol_file_name>.PRO in the VFCS working directory. All interactive commands are written (in the order of their execution) into <protocol_file_name>.PRO.

To terminate the protocolling activity and write the protocol to file, use the "Proff" button in the "Command Interpreter" menu. Exiting from VFCS with an opened protocol file will close this file. When closing the protocol file, it is possible to save the current data distribution also. The system will ask if you wish to do so in a window at the bottom of the screen. Use the "OK" to confirm, or "Cancel" button.
Illegal commands are not recorded in the protocol file. In the "prot" protocol mode, the selection of program constructs are recorded in the protocol file by a "searchstmt" <STATEMENT_intconst> command, followed by one or more "searchid" <FORTRAN_f_identifier> commands. These make use of line numbers within the program file.

For example, if the unit "test" is called in statement 200 in the active unit, the interactive selection of this subroutine call (by clicking on the name) and an interactively given "inline expansion" command will be protocolled as:

```
searchstmt  200
searchid    TEST
inlinexp
```

In the "rprot" protocol mode ("r" stands for relative positioning), the selection of program constructs is recorded differently: Positioning is recorded by

```
"top"
```

to set the cursor to the (textual) top of the unit, and by appending one or more commands

```
        "searchid"  <FORTRAN_f_identifier>
```

to jump from one occurrence of <f_identifier> to the next, until the intended occurrence has been reached. Note that this mode of recording is independent of line numbers, and thus is less likely to be affected by program modifications.

Since the protocol file is written in the same file format as a manually constructed batch file, you may modify it using a text editor. Invalid transformations and those which have no effect can be changed or deleted. Furthermore, the commands to select program constructs (recorded by "searchstmt" or "searchid" commands) may be changed, often enabling the further use of a protocol file even after a number of changes to the source program. As a consequence, minor modifications to a FORTRAN source program will not necessarily require the reconstruction of an associated batch file.

7.2 Executing Batch Files in VFCS 1.0

The commands of a batch file can be executed either by invoking VFCS in the batch mode or by selecting [**System : Batch Facility**] in the interactive mode.

7.2.1 Starting VFCS 1.0 in Batch Mode

VFCS is invoked in batch mode by typing

vfcs <batch_file_name>

VFCS will then execute the batch commands in <batch_file_name> without user interaction. Errors occurring during the processing of the batch file are displayed at the user terminal. For example:

BATCH ERROR at line 16383 : Front end terminated with error (see fe1_errfile) ! Aborting.

Note: There must be a *Units* subdirectory in the working directory before a batch file can be executed.

7.2.2 Starting Batch Files from the Interactive Mode

In the interactive mode, the batch command interpreter is activated by selecting [**System : Batch Facility**]. In the Command Interpreter window, selecting the **Verbose** mode will provide for the printing of a detailed description of the commands processed on the screen.
After clicking the "Batch" button, and setting the directory path, a list of file names with suffix "*.bat*" is displayed in the file area of the file selector box. The interpretation of the commands of a file is started by double-clicking (i.e. selecting) a file name. If the file contains macro definitions (see below), a list of names denoting defined macros is shown in the "defined macros" window. Before starting macro interpretation, the active unit, if there is one, is saved and removed from the program window.

7.3 Creating and Using Batch Macros

Macros can only be defined in batch files using the "macro" command (see Appendix B). They permit you to define a sequence of batch commands as a macro and assign a name to it. A macro can be executed using the command "exec <MACRO_name>" in the batch file where it is defined, or from the interactive system: in the Command Interpreter window, which is entered via the *Batch Facility* entry within the *System* menu, you may select a macro name and use the "Exec" or "Select macro" button to begin its execution. The available macros (which will generally be those defined in *.bat* files within the working directory) can be displayed in the "defined macros" box, or by clicking the "Show" button. By default there is no screen output for the execution of macros.
If verbose mode is selected by clicking the "Verbose" button in the Command Interpreter window, the macro execution is recorded in detail. When an error occurs, it is displayed with a brief description. The "Return" key (or the "Enter" key) aborts the execution of the macro and returns control to the window. The user is also returned to the window after successful macro execution. There are two buttons for deleting macros:

- "Clear" deletes the macro with the specified name
- "Clear All" deletes all macros currently available to the VFCS (for example, to execute once again the batch file in which these macros are defined).

There is a pre-defined macro named "fe3": this name should not be used for any macro which you create.

Chapter 8

Executing The Parallelized Code

8.1 Generating Target Code

The VFCS code generation for the iPSC/860 generates a host and a node program written in FORTRAN 77 and message-passing routines. Node-node communication and host-node communication is either implemented via appropriate communication routines supplied by Intel, e.g. CSEND and CRECV, or by subroutines generated by VFCS or contained in the VFCS libraries, e.g. SAR, RRED or WAITTG.

Code generation for computer systems on which the *Parmacs* macros are installed is similar: here, too, host and node programs are created, which are written in FORTRAN 77 and make calls to message-passing routines. Post-processing of the output is necessary.

Generated routines for the parallelized application are contained in the files *host.f*, *node.f*, *hostsys.f* and *nodesys.f*. The *init.** files are used to initialize the node codes and are thus also required at run time.

8.1.1 Creating Output Files

Code is generated via the menu items [**Generate Code : Select Target Machine : . . .**], where the machine selection initiates the appropriate code generation.
In batch mode, it may be initiated by the batch command *x_ipsc_be* for Intel iPSC/860 code, or *x_genesis_be* for Parmacs code.

The output source files are created via the menu item [**Code Generation : Save Parallel Program**] or via the batch command *final_code*.

8.2 Compiling and Executing Output Code for the iPSC/860

8.2.1 Execution with a SUN Host

If you have remote host software installed on your SUN network, you may use your SUN as the host processor, and compile and execute the output code locally. To do this, you must have the *VFCS_UTIL* variable set in your local environment. This variable referes to a directory containing files to build, compile and execute your program, so the inclusion of this directory in your local path is advantageous. Ask your system administrator for details.
Invoke the *compile* utility contained in this directory for compilation only. This creates the *host* and *node* executables. To execute your code, you need only invoke *host* from the command line. It contains a *GETCUBE*

command (and hence will cease execution at this point if the processors are not available), and includes byte-swapping routines.
Invoke the *run* utility to compile and begin execution of your code.
There are trace versions of the compile and run utilities included in the *VFCS_UTIL* directory. They are invoked by *trace_compile* and *trace_run*, respectively. These create files which may subsequently be analyzed.

8.2.2 Execution with the SRM as Host

If you are using an iPSC SRM as your host processor, the *VFCS_UTIL* variable must be set in your local environment on the SRM. This variable refers to a directory containing files to build, compile and execute your program, so the inclusion of this directory in your local path on the SRM is advantageous. Ask your system administrator for details.
To compile and execute the code on the SRM, you must first copy both the FORTRAN files generated by VFCS and the *init.** files in your working directory to a directory on the SRM. If you cannot do this by a simple *cp* or *rcp* in your environment, you should use the *ipsc_pack* and *ipsc_unpack* utilities to transfer the program. These are contained in the directory pointed to by VFCS_UTIL; naturally, you should create a script of the former on your SUN workstation.
Once you have copied the files to the the target SRM, or unpacked the tarfile via *ipsc_unpack*, you may use the *compile* utility to compile and link your code, or *run* to compile it and initiate execution.
For the SRM, there are also corresponding trace compile and run files called *trace_compile* and *trace_run*. These create files which may subsequently be analyzed.

8.2.3 Accompanying Software

The VFCS_UTIL directory contains the following files:

- ipsc_pack
- ipsc_unpack
- compile
- makefile
- run
- trace_compile
- trace_makefile
- trace_run

The compiled code makes use of the following libraries:

- for the host program: vfcs-host
- for the node program: vfcs, parti

It should not be necessary for you to link these explicitly: they are invoked in the makefiles delivered together with the system.

8.2.4 Performance Measurement and Debugging on the iPSC/860

If you do not want to use the system menus to insert statements to measure the performance of the program, then you can insert write statements in the host and the node program to follow program progress or measure its performance as described here. The local clock of the processes is called via the Intel-supplied functions *MCLOCK()* or *DCLOCK()*. The latter has a finer resolution. The node number of the executing process is provided via the function *MYNODE()*.
In addition to these basic features, the measurements can be performed by annotating the sequential source code. For this purpose VFCS provides the functions *DLTIME()* and *DMFLOP()*.

double precision FUNCTION DCLOCK()

This Intel-supplied function reads the node clock and returns the number of seconds in double precision format.

double precision FUNCTION DLTIME(time,text)

Arguments:

double precision	:	time	/* execution time */
character*(*)	:	text	/* explanation */

DLTIME prints a message on standard output which contains: Processor number, the value of *time* and the *text* as an explanation. The return value is the value of *time*.

double precision FUNCTION DMFLOP(flops,time,text)

Arguments:

integer	:	flops	/* number of operations */
double precision	:	time	/* execution time */
character*(*)	:	text	/* explanation */

DMFLOP computes the Mflop rate from the number of operations and the time. It prints the Mflop rate on standard output together with the processor number and the explanation.

Usage

If these functions are contained in the source code they are executed in the parallel program by nodes depending on the mask generated for the statement. For example a statement

time = DLTIME(time - DCLOCK()," execution time")

will be executed in each node, since *time* is a replicated variable. Thus each nodes prints the difference between time and it's local clock.
If the statement is an assignment to an element of a distributed array, such as

time(1) = DLTIME(time - DCLOCK()," execution time")

only the node which owns *time(1)* prints the execution time.
IMPORTANT: VFCS eliminates automatically dead code in the front end. Since the new functions are intrinsic functions, VFCS assumes that they do not have any side effects. Thus the statement

time = DLTIME(time - DCLOCK()," execution time")

will be eliminated unless the value of time is reused in a statement which cannot be eliminated. The easiest method to solve this problem is to change such a statement into an assigment to a replicated array. These statements are never eliminated.

8.3 Compiling and Executing Output Code with Parmacs Macros

This will be delivered separately.

8.4 Demo Codes

The system is delivered together with three demonstration programs:

- jacobi
- matrix
- mgs

Ask your system administrator where these are located in your system.

8.4.1 The Jacobi Program

This application program performs Jacobi relaxations on a two dimensional grid. The number of iterations (integer) and the relaxation parameter (a real number which should be between 0.0 and 1.0) are input parameters and you are expected to enter these at the keyboard. If you enter 0 as the number of iterations, the application is terminated. The grid size can be adapted in the code by changing the constant *nd*.

8.4.2 The MGS Program

This demo multigrid program is an abbreviated version of the mg00d. It performs V-cycles only. The delivered version is configured for a grid hierarchy of 5 levels. The finest grid thus has 33 x 33 grid points. The output of the program, i.e. the error of the solution on the finest grid and its maximal value are written to the file *ausgabe*.

8.4.3 The Matrix Demo

The simplest of the demonstration codes is the matrix multiplication. In this version, the initialization of the arrays is performed in the code and the main diagonal of the result matrix is output.

8.4.4 The Demo Files

Each of these demos is accompanied by a batch file which contains the entire transformation process in *prog.bat*, where *prog* should be replaced by the name of the demo. There is also one or more files containing a specification for the distribution of the program data (in *prog.spec*), the specification of accesses to work arrays (**.ws*) and information for the interprocedural partitioning analysis (**.adj*). These latter files are used by the batch file during its execution. The *prog.spec* files may also be read in by the interactive system to distribute the data.

You may parallelize the programs in several different ways. Make a local copy of them before you begin. Either invoke VFCS in interactive mode and enter an appropriate data distribution interactively or read in a *prog.spec* file in the data distribution window. You may be asked to supply information about the arrays passed to subroutines: this topic is discussed in Section 6.4.1.1. You may also modify the source code to specify a data distribution in Vienna Fortran syntax. Otherwise, use VFCS in batch mode to execute the batch files provided. This may be done automatically or in the following manner (see description in Chapter 7):
Start VFCS and select [**System : Batch Facility**]. Within the Command Interpreter window, select *Batch*, double click the batch file to see a list of the defined macros. To start the transformation process, execute the macro *all* from this window.

Chapter 9

Hints and Limitations

9.1 Limitations

The current system implementation is a prototype and there are currently still some minor limitations on the code it can process. These concern in particular the format of I/O statements:

1. FORTRAN 77 *READ* and *WRITE* statements must not contain:

    - io-implied-do lists
    - replicated dummy arrays
    - non-replicated adjustable dummy arrays
    - non-replicated dummy arrays with non-standard distributions
    - character variables

    The first of these, in particular, may require that you rewrite I/O statements within your program (using explicit DO loops).

2. Global reductions for the iPSC can only be used if the program is executed on an entire subcube, i.e. 2^n nodes.

3. The system rejects programs which perform no input or output whatever.

4. The system does not always correctly process I/O where array variables are individually read/written in a loop with a negative stride.

Note that **"DIST"** is a reserved word in the syntax of VFCS and may not be used as the name of a program variable or program unit.

9.2 Bugs

Several bugs are currently being processed. These include:

- The system currently does not correctly process block IF statements if the body of the statement is empty (i.e. has the form `IF(..) THEN ENDIF` without further statements).

- Some of the error messages are still in German! Error handling is do receive such a message, you may need to contact us to get help. E-mail would be the most appropriate form.

- The system does not always simplify the Boolean expressions that "mask" statements to control their execution. You may want to improve the output code in this respect.

- The system will sometimes fail to process a subroutine in which ALL data is replicated, i.e. it contains no properly distributed arrays at all. This may be resolved by introducing a new (superfluous) array which is distributed so that it has an element on each processor. However, it should also be used in the code, e.g. assigned the value 0 everywhere.

- The system will crash if an IF statement or Block IF does not have an associated statement (see above).

- The system will crash if there is a sequence of more than 12 ELSE IF clauses in a Block IF.

- A format statement which is not used in the program will remain in the code, but its label will disappear. You should remove this format statement in its entirety.

- There are several minor bugs in the "Masking" phase.

- When the system writes a protocol of a parallelization session, the statement selection does not work as intended. You will have to insert missing batch commands manually at the required positions.

Users should report bugs to us by sending the following to *vfcs@par.univie.ac.at*:

1. The source program. If possible, the relevant section of source code should be isolated and a small executable program constructed to demonstrate the problem. If this is not possible, the entire source program should be sent and/or as precise a description as possible of the error encountered, including the VFCS error message, or a description of the faulty behavior.

2. The generated code files (including *init.** files).

3. A description of the parallelization session. This can be best obtained by generating a protocol of the parallelization process.

4. A contact person who has knowledge of the program and the system on which VFCS is installed.

Please note again that VFCS does not currently provide the comfort of a compiler in its error reporting. To eliminate standard Fortran errors, run your code (without Vienna Fortran directives) through a conventional compiler.
See the Release Notes and the information on preparing your program for input to VFCS in Chapter 5 of this documentation for further information.

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

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

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

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

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

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

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

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

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

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

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

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

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

[32] 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.

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

[34] 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.

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

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

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

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

[39] 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.

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

[41] 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

[42] 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

Data Distribution Language Syntax

This appendix describes the commands of the data distribution window (5.2.2.5), including the major windows it uses to present interim results or display information to the user. In a separate section, it then presents the commands of the data distribution language. These commands may be used to describe processor array structures and to specify either a distribution of an array (including "virtual arrays") or a method by which the system should construct a distribution. These strategies are discussed in 3.2.1. There is a further command to declare an array of the program to be a workspace: the concept of a workspace is discussed in 3.2.3. If the system requires further information on the nature of actual arguments to subroutines, it will query the user. These issues are dealt with in 6.4.1.1.
When distributing an array, or requesting information on it, the array must be fully identified:

- either the name of the array must be qualified by the program unit in which it is declared, e.g. "show part difmx.u" to see distribution information on the array *U* declared in subroutine *DIFMX* (for the main program, the qualifier is always "main", whether or not the main program has a user-supplied name), or
- the program unit in which the array is declared must be active. Either it is activated via the *Units* menu prior to entering the data distribution window, or it is activated via the "load" command of the batch language. For example, to modify the partition of array *U* in subroutine *DIFMX*, the following commands will make the array available and invalidate any existing distribution:

```
load difmx
change part u
```

Note that a distribution specified by the user may be rejected by the system: this may, in particular, occur if a distributed (non-replicated) array is equivalenced with another array object, or if workspace arrays (virtual arrays) and declared arrays of the program are both used as actual arguments to the same subroutines. Information on why some distributions are considered to be illegal is given in 3.3.

The syntax below is presented in Backus-Naur form.
[...] means optional
{...} means zero or more times
| separates alternatives

Terms which are surrounded by "<" and ">" are non-terminal symbols and are either further defined, such as <distribution>, or should be replaced by actual values, e.g. for <file> you should substitute the name of the file required. Strings not enclosed in this way should be typed in as given. Single characters which must be typed in are enclosed by single quote marks, e.g. '(' indicates that the left bracket is part of the syntax of the command and should be typed in.

A.1 Data Distribution Control Commands

We first describe the commands for interactive control of data distribution which appear as buttons at the top of the data distribution window. These are also briefly described in 5.2.2.5. Recall that you must click on the input line before you can type data into it.

,

NAME batchread

SYNOPSIS batchread <file>

DESCRIPTION If you click the "Batchread" button, the file selector box will appear for you to select a file containing a data distribution for the current program's data. The default filter supplied selects those files with the suffix ".spec". If you type this command in at the input line, you must supply the name of the file.

"Batchread" reads commands of the data distribution language from the specified file (see also "read"). In contrast to "read", the commands are not presented in the input line for confirmation or modification. Input is disabled while the file is processed. You will be able to leave the window by clicking on the "Quit" button, or obtain information on the distribution, once all commands contained in the file have been executed.

,

NAME change

SYNOPSIS change option

DESCRIPTION "Change" is provided to enable you to modify an existing distribution: you may change the definition of processor array structures, modify the distribution of one or more arrays in your program and redefine a workspace.

An array or work space must be fully identified (see above): the command will invalidate the existing distribution and present it in the input window. To restore the existing distribution, simply click on "Execute". Otherwise, modify the input line as desired. If you click on "Erase line", then the array will no longer have a defined distribution!

You do not need this command to augment an existing set of distributions by new distributions or processor array structures.

The options are:

part <array_name>	change the distribution of the named array
pas <pas_name>	change the definition of an existing process address space
ws <array_name>	change a work space definition

,

NAME read

SYNOPSIS read <file>

DESCRIPTION If you click the "Read" button, the file selector box will appear for you to select a file containing a data distribution for the current program's data. There is no default filter. If you type this command in at the input line, you must supply the name of the file.

Each line in the specified file has to be a valid command of the data distribution language. Each command is read from the file and displayed in the input line. There it can be modified, if desired, and then entered by clicking on "Execute". A line may be rejected entirely by clicking on "Erase line". The data distribution window will be extended downwards when you begin executing the commands of the file: it displays information on the distributions selected for the arrays of the program. This information may be viewed for all distributions in the file via the scroll bars.

Independent of whether or not an error occurs during the execution of a command, all commands included in the file are read.

,

NAME save

SYNOPSIS save <file>

DESCRIPTION If you click on the "Save" button, you will be shown a list of files from the current directory and asked to select the name of the file to which the distribution should be saved. The existing file will be overwritten. If you enter "save" and a file name at the input line, you may create a new file or overwrite an existing file, as required. The current data distribution is saved to the given file. The file may have any name you wish; however, it will only be available subsequently for inspection or reading within the distribution window if it has a suffix known to the system. The standard forms are **.part* or **.spec*.

This option is useful if you have interactively specified a data distribution and wish to save it, or if you have changed the distribution read in from a file. The file consists of those commands of the data distribution language which will result in the construction of the same data distribution if read in via "Batchread" or "Read". It includes the "load" commands to activate the relevant program units.

,

NAME show

SYNOPSIS show <option>

DESCRIPTION "Show" displays information about the current data distribution and provides interprocedural distribution information.

If you press the "Show" button, you will have four options for the display of information. Three of them require further input before they can be processed, and thus the command will appear in the input line for your completion. Once you have entered and confirmed a complete command, the "show" window will present the required information. It must be explicitly exited. This command may, as with all others, be entered at the input line and executed from there.

The following options are recognized:

arrays Use this to obtain information on all arrays which have been given a distribution. The non-replicated arrays are displayed together with their distribution number, communication class, currently available information on overlap. This information is organized by distribution class: arrays with identical distributions have the same distribution number in VFCS.

pas <pas_name> This displays information on the specified processor array structure. The processors in the structure are numbered consecutively (i.e. linearized) and their correspondence to the linear processor array SP is displayed by showing the resulting pairs of numbers. The display currently does not reflect the neighborhood relationship of a structure. (If SP is selected, then only a list of processor numbers is given.) For example, given

```
pas p3(4,4)
pas p4(4,4) with (i=1,4 (j=1,4 p4(i,j) -> p3(j,i) ))
```

then after the processor numbers of p3 and p4 are linearized according to Fortran's column-major ordering, the following pairs of processor numbers are displayed by "show pas p4":

```
1  2  3   4  5  6   7   8  9  10  11  12  13  14  15  16
1  5  9  13  2  6  10  14  3   7  11  15   4   8  12  16
```

part <array_name> Use this option to obtain information on a specific array in the program. If the array is a "virtual array" of a work space, then the name must be qualified by the prefix *SYS*. For example, if the workspace *U* has been subdivided into arrays *G1*, *G2* and *G3*, then you may obtian information on the distribution of *G2* by invoking this command with "show part SYS.G2" or by clicking on the "Show" button, then on "Partition", and completing the input window line by adding "SYS.G2" to its contents.

This option displays detailed information about the specified array's distribution. The sement of an array mapped to a processor is shown in such a way that the parts of the array are displayed for each dimension. Overlap information is provided and neighboring processors in the communication of elements of this array is shown if relevant (and known when invoked). The maximum number of elements in a segment of the array is displayed.

ws <array_name> Use this command to see how a workspace has been subdivided into virtual arrays by means of the "ws" command. The array name must be qualified with the name of the program unit in which it was declared. For example, if array *U* is declared to be a work space in the main program, you should invoke this command with "show ws MAIN.U", or select the appropriate buttons and complete the input line by adding "MAIN.U" to its contents.

The specified array must have been declared to be a workspace via the "ws" command. Information on the declaration of the work space is provided.

In order to enter a command into the input line, or to modify a command read into the line, you must first click onto the line at the appropriate position. It may then be executed.
The following buttons appear at the bottom of the screen:

,

NAME Execute

SYNOPSIS Execute

DESCRIPTION The distribution command in the input window is ready for execution. If you press this button, the command is immediately executed and the input line cleared for the next command (or the next line of the file is read in to the input line).

,

NAME Erase line

SYNOPSIS Erase line

DESCRIPTION The distribution command in the input window should not be executed. If you press this button, the command is discarded and the input line is available for the next command (or the next line of the file is read in to the input line).

,

NAME help

SYNOPSIS	help [command]
DESCRIPTION	"Help" supplies information about the possible commands. Help with no options presents all commands.

The options are:

command If a special command is specified, this command is further explained.

This option is currently not available.

,

NAME	Quit
SYNOPSIS	Quit
DESCRIPTION	The data distribution window is closed. The rest of the system is disabled until you leave this window. You must always do this explicitly.

The following commands are not part of the distribution language proper. They must be entered at the input line or included within a data distribution file, but do not specify a distribution. They support the user in working with data distributions.

,

NAME	compute
SYNOPSIS	compute option
DESCRIPTION	"Compute" determines data partitioning information. It can be used to determine special information independently of the data partitioning transformation.

The options are:

classes	Analyzes the distributions of the partitioned arrays. The distributions are characterized by a distribution number. Distributions have the same number if they are equivalent.
callinf	The distribution vectors of the subprograms are computed.

,

NAME	end
SYNOPSIS	end
DESCRIPTION	Used within files only. It terminates the reading of data distribution information in batch processing.

,

NAME	np
SYNOPSIS	np = <intconst>
DESCRIPTION	'Np' restricts the maximal number of node processes. This restriction increases the efficiency of the analysis algorithms.

,

NAME	set

SYNOPSIS	set option
DESCRIPTION	'Set' provides the possibility to change predefined constants.
	The options are:

system	The default unit is set to be SYS.
unit	The default unit is the active unit.
eq <realconst>	The parameter needed in the equal interior blocks distribution strategy is changed. Initially it is 0.25.
gomory <intconst>	The maximal number of solution presented by the gomory algorithm during the computation of a variable distribution is set. Initially it is 24.
var	The distribution variables are set to undefined.

A.2 Data Distribution Language Commands

The following commands are used to specify data distributions for arrays in the language, declare work spaces and processor array structures.

,

NAME	load
SYNOPSIS	load <unit_name>
DESCRIPTION	The unit with the name <unit_name> is activated, or loaded. It is now possible to specify data distributions for the arrays within that unit or to request information on them. This command may be used within data distribution files or it may be entered in the input line of the data distribution window.

,

NAME	part
SYNOPSIS	part <distribution>

<distribution>	:	<array_name> <decomposition> [with <segm_map>]
<decomposition>	:	'(' <dim_part> ',' <dim_part> ')'
<dim_part>	:	<intconst> | <intconst> vf | <intconst> eq | <segm_descr_list>
<segm_descr_list>	:	<segm_descr> '-' <segm_descr>
<segm_descr>	:	[<intconst>] <length>
<length>	:	'<' <length_expr> '>'
<length_expr>	:	Such an expression may contain distribution variables, integer constants and the following operations: '*', '/', '+', '-'. The expression must be linear in the distribution variables.
<segm_map>	:	<pas_name> | <expl_segm_map>
<expl_segm_map>	:	'(' <var> '=' <intconst> ',' <intconst> [','] <expl_segm_map> ')' | <segment> '->' <process>

<segment>	:	<array_name> '(' <expr> ',' <expr> ')'
<process>	:	<pas_name> '(' <expr> ',' <expr> ')'
<expr>	:	Such an expression may contain variables defined in the segm_map, integer constants and the following operations: '*', '/', '+', '-'. The expression must be linear in the variables.
<var>	:	must not be a distribution variable

DESCRIPTION 'Part' specifies a distribution for the array. This array must belong to the active unit (i.e. loaded). The user specifies a decomposition of the array into segments and optionally a mapping of these segments onto the processors of a PAS. If no mapping is specified, the multi dimensional segment structure is linearized according to Fortran conventions and the segments are assigned to the processors with the same number.

Examples:

REAL A(100)

part a(5)

part a(5eq)

part A(2<7>-3<i>-3<j>-2<i-1>)

,

NAME pas

SYNOPSIS pas <pas_name> <pas_structure> [with <proc_map>]

<pas_structure>	:	'(' <intconst> ',' <intconst> ')'
<proc_map>	:	'(' <var> '=' <intconst> ',' <intconst> [','] <proc_map> ')' | <process> '->' <process>
<process>	:	<pas_name> '(' <expr> ',' <expr> ')'
<expr>	:	Such an expression may contain variables defined in the proc_map, integer constants and the following operations: '*', '/', '+', '-'. The expression must be linear in the variables.
<var>	:	must not be a distribution variable.

DESCRIPTION 'PAS' specifies a processor array structure. This is a multi dimensional array structure which is used as the target of array mappings. A segment of an array is mapped to a processor of a PAS. The user may supply a mapping of the elements of the PAS onto the processors of the linear processor array SP. Otherwise, a default mapping onto SP is assumed by the system. A new PAS may be mapped to an existing PAS in a variety of ways (cf. the example above for the "show" command, and in 3.2.1). An element of a PAS may be mapped to more than one element of another PAS to specify replication of data.

These structures are used to structure the mappings within a program according to the dimensionality of the data distributions selected, to specify mappings of array segments to processors other than the simple linear mappings, or to enable partial replication of array elements.

For example:

```
pas a(m,m)
pas b(m) with (i=1,m b(i) -> a(i,i) )
```

makes it possible to map arrays to the diagonal processors of the original array of processors used for an application.
The notation "->" should be thought of as "corresponds to".

,

NAME ws

SYNOPSIS ws <ws_name> <work_space_spec>

<work_space_spec>	:	<array> ',' <array>
<array>	:	<array_name> '(' <dim_list> ')'
<dim_list>	:	<dim> ',' <dim>
<dim>	:	<intconst> ':' <inconst> | <intconst>

DESCRIPTION 'Ws' specifies a work space. The ws_name must be the name of an array belonging to the active unit (loaded unit). The arrays within the work space are known as *virtual arrays* and they should be defined in the same way as ordinary Fortran arrays. The array_name may be qualified by the virtual unit SYS. It must be qualified thus when referred to, for example, when requesting information, since it is not an array declared within the program.

A.3 Selecting Solutions for Distribution Variables

Here, we explain how to select a general block distribution specified using distribution variables. This method of distribution is discussed in 3.2.1.

If you specify a general block distribution using distribution variables, the system computes solutions for the undefined distribution variables. If such solutions exist, the system presents these in a special window. The number of presented solutions is restricted, but can be changed via the 'set' command.
First, a window appear in which you are asked to confirm the starting values (referred to as *initial bounds* within the system) of the distribution variables. These bounds have to be greater than one. They can be changed by the user in the following manner. For each distribution variable defined, one or more identical bars are displayed with the name of the variable and the default initial bound. If there is more than one, previous selections of bounds for this variable in this session are recorded. To modify the bound for a specific variable, click on the (righthand) bar. It is emboldened and the lower bound appears in the input line. Click on the window in the usual way to enter the new value (you will probably want to delete the existing value first). Then click on the *OK* button to confirm the input. It should appear as the new bound on the righthand bar. Continue in this manner until you have specified lower bounds for each distribution variable which requires a value other than the default. Then confirm the selections by clicking on *OK* again. Note that these values are retained. If you use the same distribution variables in the same session, then the initial bounds will be those you have provided. It is useful to change the bounds to restrict the computation of solutions to those above a certain range if a large number of solutions are assumed to exist.
Once you have confirmed the initial bounds, the solutions to the distribution specification will be computed. These are presented in a further window.

Each solution consists of a set of values for each of the distribution variables. There are four buttons provided in the upper part of this window. The *New Bounds* button returns you to the previous window to change the initial bounds if you wish. The *Return* returns you to the data distribution window but does not make any selection from the solutions offered.

If there is more than one solution, then the first of them (generally with the highest value for the first distribution variable) will be positioned below the others, together with the distribution for the array it yields. If you now press *Take*, this solution (and hence, this distribution) will be selected. If you want to use another solution, you should enter the number of the required solution for the variables in the input area to the right above the main display. The solutions are numbered consecutively, beginning at the top left of the display, and from left to right in each row before the solutions in the next row are numbered. The solution selected will be displayed below the others, together with the array distribution it yields. Continue until you are satisfied with the distribution obtained. Make your choice by clicking on the *Take* button.

Appendix B

Batch Language

Here we define the commands which may be used to create batch files for automatic execution by VFCS in batch mode, either from the command line or from within the batch facility of the interactive system. We also show how the commands may be used by means of several simple examples.
The rules of the batch language are presented below in a BNF-like style, where <...> denotes non-terminal symbols, and | denotes a choice expression. Brackets [..] denote 0 or 1 occurrences of the enclosed expression. Braces { ... } denote 0 or more occurrences of the enclosed expression. Explanations following a double ”slash” (”//”) are not part of the batch language. They explain the semantics of the commands. Strings of terminal symbols are enclosed by quotation marks ”...”.

B.1 Constructing a Batch File

The terminology used in the batch language corresponds roughly to the labelling of menu items in the interactive system. In particular, the initial preprocessing of files before parallelization is performed in the batch language in two different ways, depending on whether the program is to be parallelized or not. There is also a set of commands for the individual parts of the front end; these should not normally be required by a user.

B.1.1 General Rules for Creating Batch Files

Only the commands defined below are legal in batch files and will be recognized by the system.
There may be only one batch command per line of a batch file.
A line beginning with a “#” is considered to be a comment line and is discarded.
If you begin batch processing from the command line, you may either specify the Fortran files which make up your source program on the command line itself or you may specify them within the file as arguments to the ”get_sources” command used.
As in interactive mode, the order in which transformations are applied does matter. If you attempt to apply transformations in an illegal order, the system will output an error message and terminate execution.

The final code must be saved explicitly. The output of the batch run, assuming it is saved, will be written to files in the directory from which you have started vfcs. The files have the same (standard) names as in interactive vfcs.
If you wish to parallelize a program with more than one output, you should do so in separate runs of vfcs and from different directories.
Batch processing may be executed in verbose mode (click on the option in the batch facility or execute with the -*v* flag from the command line). You will then receive a message whenever the batch processor begins execution of a new command, or starts working on a new program unit if it is executing a ”for_all_units” block.

Before writing a batch file, study the examples provided in the *demo* codes and at the end of this section, and generate protocols for your own programs.

B.1.2 Defaults

If there are no commands to read in the sources, the system will read them in for parallel processing by default. If you use the "z_machine" option to generate code, then code will be generated for a predefined target machine. This is currently the iPSC/860 in all installations. You should otherwise invoke the code generation for the target machine you require. There are otherwise no default assumptions made by the system.

B.2 The Commands

A list of the commands which may make up a batch file follow; a brief explanation is provided. For further information on the transformations and program manipulations, see the relevant sections of the User's Guide.

```
<batch_file>    ::= <control_block>  {  <control_block>  }

<control_block> ::=   <stmt>        // transformation statement
                  | <macrodef>        // macro definition
                  | "for_all_units"  <stmt_block>
```

The forall_units command allows you to define and execute sequences of statements for all units. The units of the program are loaded in post-order, and the sequence in < .. . > is executed for each of them. The modified units are then saved.

```
<stmt_block> ::=        "{"  <stmt>  {  <stmt>  }  "}"

<macrodef>  ::=  "macro"  <MACRO_ident>  "["  <control_block>  "]"
                                  // Macros may be executed using the exec
                                  // statement or from the interactive batch
                                  // facility.

<stmt>      ::=         <batch_stmt>            //command of the batch language
             |          <transformation>        //transformation invocation
```

The *batch_stmt* commands enable you to prepare the program for parallelizing or applying transformations, to perform the parallelization steps, to generate the output program for the target machine, to create and delete program versions, and to "load" or "select" program units (MAIN program, subroutines and functions). The *transformation* commands are individual program transformations immediately available to the user; many of them are applied to loops in the program. Naturally, other batch commands may be considered to be complex transformations, generally applied to the entire source program.

B.2.1 Batch Statements

The statements in the following are applied to the entire program. Most of them correspond 1-1 to items in the menus of the interactive tool:

```
<batch_stmt> ::=                      // major batch commands for
```

```
                                 // processing of files

// These commands: preprocessing of source files for any work with VFCS

           | "delete_old_units"   // if used, must be first command
                                 // clears unit directory of old files
           | "get_sources_par" <options>  [<filename_list>]
                                 // compiler front_end for parallelization
                                 // file names here if not on commandline
           | "get_sources_seq" <options>  [<filename_list>]
                                 // compiler front_end for sequential
                                    processing or performance profiling
                                 // file names here if not on commandline
           | "fe1_del" <options>  <filename>
                                 // call front_end_1 and delete old units
           | "fe1_nodel" <options>  <filename>
                                 // call front_end_1; old units remain
           | "fe2_seq"            // call front_end_2 without splitting
           | "fe2_par"            // call front_end_2 with splitting
           | "fe3"                // call front_end_3

// These commands: parallelizing the user program

           | "data_dist"  <filename>   // processes data distribution. For
                                   Vienna Fortran programs the filename
                                   must be ``part.vf''
                                   A user may provide data distributions
                                   in a file using the commands
                                   specified in Appendix A.
           | "start_par" <ws-file>  <adj-file>
                                 // start parallelization: with
                                    specified filenames, if given
           | "opt_comm"           // communication optimization
           | "opt_mask"           // optimization of masking
           | "compute_overlap"    // calculation of maximal overlap
                                    regions, determined over all units

// These commands: generating target code and saving the output to file

           | "gen_code_ipsc"      // call code generation for iPSC/i860
           | "gen_code_ipsc_comm" // gen code for iPSC/i860 with protocol
           | "gen_code_parmacs"   // generate code for PARMACS
           | "gen_code_suprenum"  // generate Suprenum code
           | "z_machine"          // generate code for default machine
           | "save_final_code"    // save current code version to file

// These commands: handling intermediate versions of user's code

           | "get_version"  <version_name>
                                 // load specified  version of program
                                    (this must have been saved by VFCS
```

```
                                   under the given name).
            | "set_version"  <version_name>
                                // save a version of the (transformed)
                                   source program under the given name
                                   (up to 45 characters allowed)
                                   Recall with get_version command.
            | "del_version" <version_name>
                                // delete specified program version

    // These commands: miscellaneous
            | "batch" <filename>
                                // execute all batch commands in the
                                   specified batch file, thus enter-
                                   ing a "sub-batch mode". Do not use
                                   this command recursively.
            | "version"         // show system version number (used
                                   for system compatibility check only)
            | "echo" " " " <string_of_printing_char>" " " // print string
                                   enclosed by quotation marks
            | "del_macro" <MACRO_ident> // delete macro with specified name
            | "exec" <MACRO_ident> // invoke a defined macro
```

B.2.2 Commands for Selecting and Saving Units

The batch commands which follow are used to load (activate) and store units, and to select program constructs by means of the positioning commands, which set an (invisible) cursor to a line, subroutine call or DO-loop.

In the batch language, you must load a unit to activate it: some batch commands are applied to the entire program, and others are applied to a program unit or to a statement or construct (e.g. DO loop) within a unit. You must always load a unit explicitly using the commands provided before you apply a transformation to that unit or select a program construct within the unit for transformation. It is not necessary to specifically deactivate a unit: saving a unit will recreate source code and write this to file, either under a generic name or as specified. Two forms are provided; for the former, the unit must be in memory and the code is written to a file simply called "fortran".

```
    // These commands: activating and saving units
            | "select_unit" <unit_name>    // activate unit with given name
            | "load" <unit_name>    // load (activate) unit with given name
            | "save_active_unit"    // save active unit
            | "save_unit"  <unit_name>  <filename>
                                 // reconstruct specified unit (from
                                    the internal representation) into
                                    a FORTRAN source file and save it
                                    under the specified filename
```

B.2.3 Commands for Selecting Statements in a Unit

The following commands enable you to specify statements within a selected program unit so that transformations may be applied selectively to them. They enable you to search for the i'th occurrence of a program variable or pick a statement with a specified Fortran label.

The following loop transformations require you to select a program variable in the code:

```
inline_exp
```

The following loop transformations require you to select a statement in the code:

```
scalar_fwd
loop_dist
loop_unroll
loop_unroll_jam
loop_stripmine
propagate_mask_stmt
```

The first part of the name indicates whether the statement must be a DO loop header.
If you select an occurrence of a program variable (but NOT a DO loop variable, see below), then the statement it occurs in is also selected. Hence both forms of selection may be used here.

The following loop transformations require you to select a pair of statements in the source code:

```
loop_interchg
```

To do so, select and mark the first loop and then select the second of the two loops.

There are two pairs of search statements defined below, one of which should be used by the system only, for the following reason. Transformations applied by the system will modify the names of DO loop variables and sometimes also modify the DO loop label. If you search for those in your original code, the batch processor may fail to find them and hence terminate with an error message.
You should thus search for statements by searching for program variables and statement labels only. Note that optimizations also eliminate and move statement labels in the original source code. So that you are able to identify specific statements in the source code for the correct application of transformations, the system will, however, remember where you have positioned statement labels. Hence, if you cannot identify a statement by the occurrence of a variable (distinct from the DO loop variable), you should prefix a label to the required statement and use the "search_id" command to find it.

```
    // These commands: selecting statement, subroutine call or loop
       by appropiately positioning an invisible cursor


                | "top"                 // reset cursor  to the (textual)
                                           beginning of the active unit
                | "mark_stmt"           // save cursor position, thus
                                           "marking" the selected statement
                | "go_to_mark"           // reset cursor to "marked" statement
                                            in active unit; the mark is lost
                                            if the active unit changes
                | "search_stmt" <statement_number>
                                        // all statements are numbered,
                                           starting at the beginning of the
                                           unit in textual order. This
                                           command sets the cursor on the
                                           statement with the specified number.
                | "+" <statement_offset> // position cursor on statement with
```

```
                              number = number of selected
                              statement + offset
      | "search_do" <LABEL_number>
                           // position cursor on DO loop with
                              specified label (this option for
                              system use only)
      | "search_do"  <identifier>
                           // set cursor on DO loop with
                              the specified loop variable
                              (this option for system usage only)
      | "search_id" <LABEL_number>
                           // set cursor on statement
                              labelled with <LABEL_number>
                              (labels are unique in FORTRAN
                              subprograms)
      | "search_id" <identifier>
                           // set cursor on next (textual
                              following) occurrence of the
                              specified identifier, starting
                              from current cursor position.
```

B.2.4 Transformation Commands

The transformations which may be applied by the user in interactive VFCS, whether to the entire program, to a complete program unit, or to a loop or statement within the active unit, are also available in batch mode. The transformation commands available to the user are the following:

```
<transform> ::=
    // These commands: transformations on all units

                 "const_prop"         // constant propagation

    // These commands: transformations on or within a program unit.
       A unit must be active (i.e. loaded) before they are applied.


                 "dead_code"          // elimination of useless code
               | "expr_simplify"      // expression simplifier
               | "gen_vec_code"       // generation of vector code
               | "induction_var"      // induction variable substitution
               | "inline_exp"         // inline expansion
               | "loop_dist"          // loop distribution (for selected loop)
               | "loop_dist_all"      // loop distribution (for all loops)
               | "loop_interchg"      // loop interchange (exchange)
               | "loop_unroll" <depth> // loop unrolling : arg is unroll depth
               | "loop_unroll_jam" <depth>  // loop unrolling and jamming
               | "loop_stripmine" <factor> // stripmine with given factor
               | "propagate_mask_stmt" // mask propagation for a marked stmt
               | "reduction"          // mark transformable vector
                                         reductions DOTPRODUCT, SUM,
                                         PRODUCT, ANY, ALL, MINVAL and
```

```
                                  MAXVAL
             | "scalar_fwd"          // scalar forward substitution
             | "unit_overlap"        // calculation of overlap regions
```

In the specifications above, a number of constructs have been used. They are defined below:

```
<filename_list>    ::= <filename> { <filename> } // list of source files
<filename>         ::= {UNIXpath}<Fortran_filename> // name of source file, with
                                            path if needed (up to 512 chars)
<Fortran_filename> ::= <identifier>.f          // name of Fortran 77, Vienna
                     | <identifier>.vf         // Fortran or HPF source
                     | <identifier>.hpf
<version_name>     ::= <identifier>        // identifier of intermediate
                                              version; up to 45 characters
<unit_name>        ::= <identifier>        // unit identifier; up to 45
                                              characters
<MACRO_ident>      ::= <identifier>        // macro identifier; up to 40
                                                 characters

<identifier>       ::= a valid Fortran 90 identifier

<LABEL_number>     ::= "0"  ..  "999999"       // FORTRAN label

<statement_offset> ::= "0"  ..  "999999"       // non-negative integer
<statement_number> ::= "1"  ..  "999999"       // positive integer

<string_of_printing_char> ::=  { <PRINTING_character> }
                                           // string printed with "echo"

<PRINTING_character> ::= "A"  ..  "Z"
                        |"a"  ..  "z"
                        |"0"  ..  "9"
                        |" "  ..  "/"
                        |":"  ..  "?"

<options>          ::= "-"  {  <opt>  }

<opt>              ::= "s"  | "c" | "i" | "l" | "f" | "e" | "o"
                       // The default settings should be used except in
                          exceptional circumstances. For a description of
                          these, see front end options in interactive VFCS.
```

Note: The batch language does not distinguish between upper and lower case letters.

B.3 Example Batch Files

In this section we present two valid batch files which can be processed by VFCS in batch mode.

B.3.1 A Standard Batch File

The following simple example shows a batch file for parallelization of a Vienna Fortran source program. The source files are specified on the command line, and the default preprocessing for parallelization is performed. Note the order in which the parallelization transformations are applied.

```
# This is a simple parallelization run: it can be applied to any
# Vienna Fortran program
# source files are read from command line; now process data distribution
      data_dist part.vf
        for_all_units
# These transformations are applied to each program unit in the entire
# code
        {
           induction_var
           reduction
        }
# Parallelize program
        start_par
        opt_comm
        opt_mask
#Generate target parallel code with calls to parmacs routines
        gen_code_parmacs
#save this code to files for execution on target machine
        save_final_code
```

B.3.2 A Customized Batch File

The next example shows the definition of several customized macros. This batch file is constructed to parallelize a specification application program. It reads in directive files with user information on the use of workspaces and the shape of array arguments and selectively applies specific transformations. The macros may be invoked from within this (or another) batch file, or they may be started from the batch processing facility in the interactive VFCS tool.

```
# This macro also assumes the default preprocessing of source files for
# parallelization. Distribution information is read from user-supplied files.

macro start [
      data_dist myinfo.spec
      start_par mywsinfo.ws myarginfo.adj
    ]

#This macro prescribes the transformations applied to two program units and
#then the standard optimizations for improving the communication and spreading
#of DO loops. It assumes that initial parallelization has been performed.

macro opt [
#This searches for reductions in the program unit myunit1
      select_unit myunit1
      reduction
      select_unit myunit2
```

```
#This searches for the do loop with statement(!) label 140, marks it
      search_id 140
      mark_stmt
#and then for the do loop with statement(!) label 150
      search_do 150
#and exchanges the two loops.
      loop_interchg
#The cursor is now positioned at the first statement in the program unit
      top
      search_id iabj
#The second occurrence of the program variable iabj is selected
      search_id iabj
#and scalar forward substitution applied to it.
      scalar_fwd
#After individual transformations, overlap information should be recomputed
#before optimizations are applied, to improve their analysis.
      expr_simplify
      compute_overlap
#Apply standard optimizations to optimize communication and striping of loops
      opt_comm
      opt_mask
  ]

#This macro generates code for the iPSC/860 and saves it to file without
#further processing.

macro end [
      gen_code_ipsc
      save_final_code
    ]

#This macro executes several macros which have already been defined

macro all [
      exec start
      exec opt
      exec end
    ]

#If the next line is included in the source file, it can be invoked from
#the command line to perform the transformations specified in macro all
#(which itself invokes further macros). Otherwise, the file defines macros
#only. They are then read in and executed via the batch facility of the
#System menu in the interactive VFCS.

 exec all
```

Appendix C

Vienna Fortran Syntax

C.1 Syntax Metalanguage

The syntax of the language extensions is specified in a variation of Backus–Naur form (BNF). We use the following conventions:

1. Nonterminal symbols are written as lower-case words (often hyphenated and abbreviated). Nonterminal symbols of the FORTRAN 77 standard are written in *italic* and have the same meaning as in the standard.
2. Keywords are written in boldface, for example **REAL**.
3. Strings of terminal symbols that are not keywords are enclosed in quotes: for example "(".
4. The following syntactic meta symbols are used ("xyz" stands for any legal syntactic class phrase):
   - → introduces a syntactic class definition
   - | introduces a syntactic class alternative
   - [ ] encloses an optional item
   - () encloses an item which specifies a set of alternatives
   - [xyz]... expresses repetition of xyz (0 or more times)
   - xyz ... expresses repetition of xyz (1 or more times)
5. In order to minimize the number of syntax rules and to convey appropriate context information, the following rules are assumed:
   - xyz-list → xyz [“,” xyz]...
   - xyz-name → *name*
   - integer-xyz → xyz

C.2 Intrinsic Functions

We will use the following notation:

- *int-array-constant* Constraint: integer array with index domain $[1 : N]$ for some N. It has to be defined via the parameter statement. (e.g. PARAMETER (BB = (/10,30,70,85/))
- *array* arbitrary array
- *element* data element (scalar or array element)

C.2.1 BLOCK

Form: *BLOCK*

Result type: 1-1 distribution function
Result: Creates a block distribution. See Section 3.2.1.

C.2.2 B_BLOCK

Form: *B_BLOCK(int-array-constant)*

Result type: 1-1 distribution function
Result: Creates a general block distribution. Let

- $[L : U]$ the index domain associated with the array (dimension) to be distributed.
- M the number of processors in the processor array (dimension), which is the target of the distribution. Then $N \geq M - 1$.
- The array (dimension) is partitioned into M contiguous blocks. For all $i, 1 \leq i < M$, *int-array-constant*(i) specifies the upper bound of Block i. The index ranges associated with the blocks are given as follows: Block 1: $[L : \textit{int-array}(1)]$ Block i, $1 < i < M$: $[\textit{int-array}(i-1)+1 : \textit{int-array}(i)]$ Block M: $[\textit{int-array}(M-1)+1 : U]$ The values of *int-array* must be defined in such a way that each block is associated with a nonempty index range.

C.2.3 S_BLOCK

Form: *S_BLOCK(int-array-constant)*

Result type: 1-1 distribution function
Result: Creates a general block distribution. Let

- $[L : U]$ the index domain associated with the array (dimension) to be distributed.
- M the number of processors in the processor array (dimension), which is the target of the distribution. Then $N \geq M - 1$.
- The array (dimension) is partitioned into M contiguous blocks. For all $i, 1 \leq i < M$, *int-array-constant*(i) specifies the size of Block i. The index ranges associated with the blocks are given as follows: Block 1: $[L : L + \textit{int-array}(1) - 1]$ Block 2: $[L + \textit{int-array}(1) : L + \textit{int-array}(1) + \textit{int-array}(2) - 1]$ ... Block M: $[\ldots : U]$

The values of *int-array* must be specified in such a way that all index ranges are nonempty.

C.3 Syntax

C.3.1 Basic Elements

1. array-section $\rightarrow$ *array_name* ["("section-subscript-list")"]
2. section-subscript $\rightarrow$ subscript | subscript-triplet
3. subscript $\rightarrow$ *int_constant_expr*
4. subscript-triplet $\rightarrow$ [subscript] ":" [subscript] [":" stride]

5. stride $\rightarrow$ *int_constant_expr*
6. declaration-annotation $\rightarrow$ actual-array-annotation | dummy-array-annotation
7. actual-array-annotation $\rightarrow$ static-array-annotation
8. extension-executable-statement $\rightarrow$ distribute-statement | forall-loop
9. int-array-constant $\rightarrow$ array_constant_name | "(/" int_constant_expr-list "/)"

C.3.2 Processor Declarations

1. processor-declaration $\rightarrow$ primary-processor-structure [secondary-processor-structures]
2. primary-processor-structure $\rightarrow$ Constraint: PROCESSORS
   array-declarator
3. secondary-processor-structures $\rightarrow$ Constraint: RESHAPE
   array-declarator–list

C.3.3 Processor References

1. processor-reference $\rightarrow$ processor-element-name | processor-section ["(/"dimension-permutation"/)"]
2. processor-element-name $\rightarrow$ *array_element-name*
3. processor-section $\rightarrow$ array-section
4. dimension-permutation $\rightarrow$ *int_constant_expr*–list

C.3.4 Distribution Expressions

1. distribution-expression $\rightarrow$ simple-distribution-expression | composite-distribution-expression
2. simple-distribution-expression $\rightarrow$ distribution-extraction
3. distribution-extraction $\rightarrow$ "=" array-or-dimension
4. array-or-dimension $\rightarrow$ *array_name* | array-dimension
5. array-dimension $\rightarrow$ *array_name* dimension-qualifier
6. dimension-qualifier $\rightarrow$ "." *int_constant_expr*
7. composite-distribution-expression $\rightarrow$ dimensional-expression-list
8. dimensional-expression $\rightarrow$ simple-distribution-expression | ":"

C.3.5 Static Array Annotations

1. static-array-annotation $\rightarrow$ [distribution-specification]
2. distribution-specification $\rightarrow$ Constraint: DIST
   dspec
3. dspec $\rightarrow$ "("distribution-expression")" [Constraint: TO
   processor-reference] | Constraint: TO
   processor-reference

C.3.6 Procedures

1. dummy-array-annotation → inherit-annotation
2. inherit-annotation → Constraint: DIST
   "(" "*" ")" ["," distribution-range]

C.3.7 FORALL Loops

1. forall-loop → forall-statement private-var-decls forall-block end-forall
2. forall-statement → label-forall-statement
3. label-forall-statement → Constraint: FORALL
   label forall-control
4. forall-control → control-variable [on-clause]
5. control-variable → *variable_name* "=" *integer_expr* "," *integer_expr* ["," *integer_expr*]
6. on-clause → Constraint: ON
   processor-element-name | Constraint: ON
   processor-reference
7. processor-reference → Constraint: OWNER
   "(" *array_element_name* ")"
8. private-var-decls → *dimension_statement* | *type_statement*
9. forall-block → reduction-statement | *executable_statement*
10. end-forall → *continue_statement*
11. reduction-statement → Constraint: REDUCE
    "(" reduction-op "," *variable* "," *expression* ["," order] ")"
12. reduction-op → Constraint: SUM
    | Constraint: MULT
    | Constraint: MAX
    | Constraint: MIN
    | *function_name*
13. order → Constraint: LEFT
    | Constraint: RIGHT
    | Constraint: TREE

C.4 Constraints

- The number of processors has to be fixed at compile time.
- Actual argument arrays must not be array sections.
- Procedure local processor arrays must not be declared. (The procesor arrays of the main program are visible in all subroutines.)

- The distribution of dynamic arrays must be known at every program point.

Appendix D

Vienna Fortran Applications

In this section, we show how Vienna Fortran can be used to express scientific algorithms. In particular, we present three examples: Gaussian Elimination, ADI iteration, and a sweep over an unstructured mesh. These examples demonstrate the flexibility and versatility of the language.

D.1 Gaussian Elimination

The Gaussian elimination algorithm is a frequently used method to solve a set of linear equations. It has been studied extensively and optimized forms of the algorithm are included in some of the major numerical libraries. Figures D.1, D.2, and D.3, present a version of the algorithm expressed in Vienna Fortran. The code reproduced here has not been written as a library routine, but is a complete program to find the solution to a set of equations. The matrix of coefficients for the set of equations to be solved is contained in the array A while B is the right hand side. Performance studies of this algorithm on various parallel machines have indicated that a cyclic column distribution for the matrix A frequently leads to a better overall performance than a block or cyclic row distribution, and hence is often the preferred choice for its distribution (although a carefully chosen user-defined distribution using blocks or a two-dimensional processor array and a cyclic distribution in both dimensions of A may be superior in some cases (cf. [36])).
Hence, as shown in Figure D.1, the second dimension of A is cyclically distributed across all processors. The array B is distributed by aligning it with the second dimension of array A; this has the effect of distributing the elements of B in a round-robin fashion to the processors. The other arrays in the program have the same distribution, and hence are aligned with B. We could equally well have aligned them with the second dimension of A or specified a cyclic distribution directly. An advantage of the strategy used here is that, if we want to test the behavior of this algorithm under different distributions, we need to modify at most the distribution annotations for arrays A and B. The alignment of the other arrays with B do not need to be changed.
The program starts by reading in the arrays A and B from a conventional formatted files in the normal manner. All standard FORTRAN 77 file operations are supported by Vienna Fortran. However, the language also supports special concurrent file operations to open close, read, write and manipulate concurrent files. In this program we want to store the results in a concurrent file which must thus be opened by a **COPEN** statement. This statement takes the same arguments as the FORTRAN 77 **OPEN** statement, but it may be used to open existing files only if they were written with the corresponding concurrent write statement **CWRITE**. The concurrent file containing the program's results may be subsequently read in by another program using the **CREAD** statement. The specification **SYSTEM** used here indicates that the array is stored in the file system using a default distribution (which may be different from the one used in the program).
The first subroutine *FGAUSD*, shown in Figure D.2, has the task of decomposing the matrix A. We have chosen to modify the sequential algorithm by expanding the temporary variable used into an array *TEMP*; thus each processor has a local variable for the local columns of A, eliminating unnecessary communication. Some compilers will be able to recognize that this is being used as a local variable and perform this transformation

```
      PROGRAM   Gauss
      PARAMETER  (N = 4000)

      REAL  A(N,N)  DIST  (:, CYCLIC )
      REAL  B(N), TEMP(N)   DIST (=A.2)
      INTEGER  IPIVOT(N)   DIST  (=B)
      LOGICAL  SING(N)       DIST  (=B)

      COPEN ( UNIT  = 6, FILE  = '/CFS/MM/GAUSS/SOL')
      OPEN ( UNIT  = 7,  FILE  = '/USR/MM/GAUSS/MAT')

C     Read data from conventional files

      READ (7,2100) ( (A(I,J), J = 1,N), I = 1,N))
      READ (7,2100) ( B(I), I = 1,N)

      DO  10 I = 1, N
10        SING(I) =   .FALSE.


C     Perform matrix decomposition

      CALL  FGAUSD(N,A,IPIVOT, SING, TEMP)


C     Test for singularity; perform solution step if matrix is not singular

      IF  (  ANY (SING) )  THEN
           PRINT  2000
      ELSE
           CALL  FGAUSS(N,A,IPIVOT,B, TEMP)
C         Write solution to concurrent file
           CWRITE (6, SYSTEM ) B
      END IF

2000  FORMAT (22H SINGULARITY IN MATRIX)
2100  FORMAT ( F8.3 )
      STOP
      END
```

Figure D.1: Program for Gaussian Elimination

```
      SUBROUTINE  FGAUSD(N,A,IPIVOT,SING,TEMP)

C    Distributions are inherited from the calling routine
      REAL  A(N,N)             DIST (*)
      REAL  TEMP(N)          DIST (=A.2)
      INTEGER  IPIVOT(N)  DIST (=A.2)
      LOGICAL  SING(N)      DIST (=A.2)

      DO   30 K = 1, N-1

 C      Find K'th pivot index, store in IPIVOT(K)
          IPIVOT(K) = K
          DO   40 I = K+1, N
                IF  (A(I,K)  .GT.  A(IPIVOT(K),K)) IPIVOT(K) = I
40       CONTINUE

          TEMP(K) =   A(IPIVOT(K), K)
           IF  (TEMP(K)  .EQ.  0.0)  GOTO  200
          A(IPIVOT(K),K) = A(K,K)
          A(K,K) = TEMP(K)

 C      Find scaling factors
          TEMP(K) = -1.0 / A(K,K)
           DO   50 I = K+1, N
50              A(I,K) = TEMP(K) * A(I,K)
           DO  60 J = K+1, N
               TEMP(J) = A(IPIVOT(K),J)
               A(IPIVOT(K),J) = A(K,J)
               A(K,J) = TEMP(J)
                     DO   60 I = K+1, N
                         A(I,J) = A(I,J) + A(IPIVOT(K), J) * A(I,K)
60        CONTINUE
30   CONTINUE

     GOTO  300
200  SING(K) =  .TRUE.
300  IPIVOT(N) = N
     RETURN
     END
```

Figure D.2: Subroutine for matrix decomposition

```
      SUBROUTINE  FGAUSS(N,A,B,IPIVOT,TEMP)

      REAL  A(N,N)              DIST (*)
      REAL  B(N)                 DIST (=A.2)
      REAL  TEMP(N)          DIST (=A.2)
      INTEGER  IPIVOT(N)  DIST (=A.2)

      DO   10 K = 1, N-1
          TEMP(K) = B(IPIVOT(K))
          B(IPIVOT(K)) = B(K)
          B(K) = TEMP(K)
          DO   10 I = K+1, N
               B(I) = B(I)  + TEMP(K) * A(I,K)
10    CONTINUE

      DO   20 K = N, 1
          B(K) = B(K) / A(K,K)
               DO   20 I =  1, K-1
               B(I) = B(I) - B(K) * A(I,K)
20    CONTINUE

      RETURN
      END
```

Figure D.3: Subroutine for the solution step of Gaussian Elimination

automatically. The DO-loop with loop variable J can be executed in parallel in all iterations.
We do not want to redistribute the arrays in the subroutine, and specify this by using **DIST(*)**. We could have annotated each of the declarations with **DIST(*)** also. However, the alignments given explicitly are equivalent to the distributions in the main program, so no redistribution takes place. If a subroutine is separately compiled, then it is advantageous to explicitly specify any alignments which are to hold, as we have done here, even if the user knows that redistribution will not take place.
The singularity test in the main program uses the function ANY, which returns the value **.TRUE.** iff any of the elements of its argument are true. If no singularities are found, execution proceeds with a call to the subroutine $FGAUSS$, shown in Figure D.3, where the solution step is performed. Here too, all arrays are used in their original distribution.
Note that only a few changes were made to the sequential program to obtain the above parallel program, the major issue being the specification of data distribution. Thus, it is easy to experiment with different distributions by just changing the declarations and recompiling.
Even though the subroutines inherit the distributions of the arguments, the presumption that the array A is distributed only in the second dimension is built into the code. This assumption may not be appropriate if the algorithm is written as a library routine, rather than a subroutine in a user program. By utilizing the intrinsic distribution query functions and the **SELECT DCASE** statement, the routines can be transformed into a version which can accept a wide range of distributions. For example, the distribution of A may determine the most appropriate algorithm for obtaining the pivot element.

```
      PARAMETER (NX = 100)
      PARAMETER (NY = 100)


      REAL  U(NX, NY)  DIST  (:,BLOCK)
      REAL  F(NX, NY)  DIST  (:,BLOCK)

      REAL  V(NX, NY)  DYNAMIC ,  RANGE ( (:,BLOCK), (BLOCK, :)), DIST  (:,BLOCK)

      CALL  RESID( V, U, F, NX, NY)


C     Sweep over x-lines

      DO  10 J = 1, NY
           CALL  TRIDIAG( V(:, J), NX)
10     CONTINUE

      DISTRIBUTE  V :: ( BLOCK , : )


C     Sweep over y-lines

      DO  20 I = 1, NX
           CALL  TRIDIAG( V(I, :), NY)
20     CONTINUE


      DO  30 J = 1, NY
           DO  30 I = 1, NX
               U(I, J) = V(I, J)
30     CONTINUE
```

Figure D.4: An ADI iteration

D.2 ADI Iteration

ADI (Alternating Direction Implicit) is a well known and effective method for solving partial differential equations in two or more dimensions [25]. It is widely used in computational fluid dynamics, and other areas of computational physics. The name ADI derives from the fact that "implicit" equations, usually tridiagonal systems, are solved in both the x and y directions at each step. In terms of data structure access, one step of the algorithm can be described as follows: an operation (a tridiagonal solve here) is performed independently on each x-line of the array followed by the same operation being performed, again independently, on each y-line of the array.

The code for such a step of the ADI algorithm is shown in Figure D.4. Here, the arrays U and F, the current solution and the right hand sides respectively, are distributed such that the columns are blocked over the implicit one-dimensional array of processors, **$P**.

The array V, used as a workarray, is declared to be dynamic with the *range attribute* specifying that the only distributions allowed are blocking by rows or columns. The first loop ranges over the columns (representing the x-lines), calling a subroutine $TRIDIAG$ for each column of V while the second loop ranges over the rows (representing the y-lines). Here, the subroutine *TRIDIAG* is given a right hand side and overwrites it with the solution of a constant coefficient tridiagonal system.

In this version of the algorithm, the array V is dynamically redistributed in between the two loops; in the first loop it is blocked by columns while in the second it is blocked by rows. Thus, in each loop, we can employ a sequential tridiagonal since neither x-lines in the first loop nor the y-lines in the second loop cross processor boundaries. Note that the redistribution of the array is a "transpose" of the array with respect to the set of processors and requires each processor to exchange data with each of the other processors. Hence, all the communication in this version of the algorithm is contained in the redistribution while the tridiagonal solves run without interprocessor communication. The final assignment of the array V to the array U also induces communication similar to the "transpose" above since U ad V are distributed in different dimensions.

The version of ADI here is only one of a number of ways of encoding the algorithm. For example, one could leave the array V in place and employ a parallel tridiagonal solver in the second loop. This would shift the interprocessor communication in the algorithm from the redistribution (and the final assignment) to the tridiagonal solvers. Similarly, the arrays could be blocked in both the dimensions and a parallel tridiagonal solver used for both the x- and the y-lines.

All versions of this algorithm are equally easy to express in Vienna Fortran. Moreover, it is a trivial matter to change the distributions, or to substitute the calls to the sequential tridiagonal solver used here by calls to a parallel tridiagonal solver. In marked contrast, such changes will typically induce weeks of reprogramming in a message-passing language.

D.3 Sweep over an Unstructured Mesh

Several scientific codes are characterized by the fact that information necessary for effective mapping of the data structures is not available until runtime. Examples of such codes include but are not limited to, particle-in-cell methods, sparse linear algebra, and PDE solvers using unstructured and/or adaptive meshes.

In this section, we consider a "relaxation" operation on an unstructured mesh. As shown in Figure D.5, such meshes are generally represented using adjacency lists which denote the neighbors of a particular node of the mesh. Thus, *NNBR(i)* represents the number of neighbors of node i while *NBR(i, j)* represents the jth neighbor of node i. The relaxation operation, as shown here, consists of determining a new value of the array U at each point in the grid, based on some weighted average of its neighbors.

In the code, the primary array *NBR* is explicitly distributed via the **INDIRECT** distribution mechanism. The distribution of *NBR* is determined by the mapping array *MAP*, which is defined in the routine *PARTITION* based on the structure of the mesh. The secondary arrays, *NNBR*, *U*, *UTMP*, and *COEF*, are automatically distributed according on the alignments specified in the respective declarations. The *notransfer attribute* specifies that the values of the array *UTMP* need not be moved when the array is redistributed.

The rest of the code depicts K sweeps over the unstructured mesh. The important point here is that to access the values at neighboring nodes, the elements of the vector *UTMP* are indexed by the array *NBR*. Given that *NBR* is distributed at runtime, the compiler does not have enough information at compile-time to determine which of the references are non-local. In such situations, runtime techniques as developed in [17, 18, 33] are needed to generate and exploit the communication pattern.

```
      PARAMETER (NNODE = 1000)
      PARAMETER (MAXNBR = 12)

      INTEGER  NBR(NNODE, MAXNBR)  DYNAMIC,   DIST (BLOCK, :)
      INTEGER  NNBR(NNODE)  DYNAMIC,   CONNECT  (=NBR.1)
      REAL  U(NNODE)     DYNAMIC,   CONNECT  (=NBR.1)
      REAL  UTMP(NNODE)  DYNAMIC,   CONNECT  (=NBR.1)
      REAL  COEF(NNODE, MAXNBR)  DYNAMIC,   CONNECT  (=NBR)
      INTEGER  MAP(NNODE)  DIST (BLOCK)

C     Define the array MAP to partition the mesh based on its structure.

      CALL  PARTITION( NBR, NNBR, MAP )

C      Redistribute the array NBR based on the array MAP. Arrays NNBR, U, UTMP
C      and COEF are automatically redistributed. The values of UTMP are not
C      transferred.

      DISTRIBUTE  NBR :: (INDIRECT(MAP),:)  NOTRANSFER (UTMP)

      DO  10 ITER = 1, K

C        Copy the values of U into UTMP
           DO  20 I = 1, NNODE
               UTMP(I) = U(I)
20        CONTINUE

C        Sweep over the mesh.

           DO  30 I = 1, NNODE
               T = 0.0
               DO  40 J = 1, NBR(I)
                    T  = T + COEF(I, J) * UTMP( NBR(I, J) )
40             CONTINUE
               U(I) = U(I) + T
30         CONTINUE
10     CONTINUE
```

Figure D.5: Relaxation sweep over an unstructured mesh

Appendix E

Installing VFCS

This release of VFCS can be installed on any SUN SPARCstation under OS4.0 or higher. Binary versions may be available for other workstations: consult the system administatrator at the Computer Science Department of the University of Vienna or send an e-mail message to *vfcs@par.univie.ac.at* for further information.

This version requires the following:

- OS 4.0 or higher
- 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 the Parmacs macros (Version 5.0 or above) have been installed, using the Parmacs macros.

The system is available both on tape and as a tar file via ftp by arrangement. To install VFCS, please enter the target directory and extract the files with the command:

- from tape: **tar xf /dev/rst8**
- from the tar file: **tar xf vfcs.tar**

This will create the following files and subdirectories:

- the binary files be1, be_genesis, be_ipsc, be_paragon, fe1, fe2, recon, spec, and vfcs. Do not delete any of these files under any circumstances.
- the file *Vfcsdefaults*. This should be copied to the home directory of each user of VFCS.
- the subdirectory *demo*, which contains source code for several demonstration programs. Three of these are further described in Chapter 8.
- the subdirectory *sun_util*. This contains utilities and libraries which are required for the execution of VFCS output on the Intel iPSC/860, if a SUN SPARCstation is to serve as the host processor. Their installation is described below.

- the subdirectory *parmacs_util*, which contains utilities and libraries which are required for the execution of VFCS output on a machine on which the Parmacs routines are installed.

- There is also a file ipsc.tar with which you may create the subdirectory *ipsc_util*. This contains utilities and libraries which are required for the execution of VFCS output on the Intel iPSC/860, if the SRM of that machine is to serve as the host processor. Their installation is described below.

E.1 Ipsc_Util and Sun_Util

These directories both contain two subdirectories. The *bin* subdirectory contains several routines to enable transfer of files, compilation and execution of the VFCS output. The *lib* subdirectory contains three library files. The file *libvfcs-host.a* is required by the host program at run time and the libraries *libvfcs.a* and *libparti.a* are required by the node program. These should be copied to the standard library paths in the local network. This is usually as follows:

- For compilation and execution on an iPSC/860 SRM as host processor:
  - Copy the file *ipsc_util/lib/libvfcs-host.a* to */usr/lib* on each SRM which will compile and execute VFCS output.
  - Copy the node libraries *libvfcs.a* and *libparti.a* in *ipsc_util/lib* to */usr/ipsc/XDEV/i860/lib-coff* on the SRM.
- For compilation and execution on a SUN SPARCstation as host processor:
  - Copy the file *sun_util/lib/libvfcs-host.a* to */usr/lib* on the SUN network and execute *ranlib libvfcs-host.a*.
  - Copy the node libraries *libvfcs.a* and *libparti.a* in *sun_util/lib* to *$IPSC_XDEV/i860/lib-coff* in a SUN network on which the Intel remote host software has been installed.

The utility files in *bin* should be copied to a convenient place in your system where they may be accessed by all users of VFCS.

- For compilation and execution on the SRM as host processor: Copy the contents of directory *ipsc_util/bin* to a convenient place on the SRM(s) where the compiled code is to be executed. Set the environment variable VFCS_UTIL to this directory (the *utilsdir*) for each VFCS user on the SRM, so that it may be correctly referenced (see below). Users may wish to include the directory VFCS_UTIL in their execution paths.

- For compilation and execution on a SUN SPARCstation as host processor: Copy the contents of directory *sun_util/bin* to a convenient place on the local network of SUNs where remote host software has been installed. Set the environment variable VFCS_UTIL to this directory (the *utilsdir*) for each user of the system, so that it may be correctly referenced. Users may wish to include the directory VFCS_UTIL in their execution paths.

E.2 Parmacs Utilities

The directory *parmacs_util* contains two subdirectories:

- *bin*, containing utilities for postprocessing VFCS-Parmacs output

- *lib*, containing utilities for macro expansion, compiling, and executing programs on the target machine.

Copy the following files from these directories to an appropriate directory on the host processor of the target machine and set the value of the environment variable PARMACS_UTIL to that directory in the host processor's environment:

- $PARMACS_UTIL/bin
  - compile_parmacs
  - parmacs_unpack
  - std_makefile

  Furthermore, the Parmacs macro definitions should be supplied in a file named *f.m4.monmacs*.

- $PARMACS_UTIL/lib
  - compile (a shell script for expanding and compiling the Fortran program)
  - Parmacs (a directory containing a shell script for installing the Parmacs macros)

  Furthermore, the Parmacs subdirectory should contain the Parmacs macro definitions.

The following environment variable is used within the shell scripts of the Parmacs utilities:

- PARMACS_UTIL : this should be set in the .cshrc file of each user, and the execution path should include $PARMACS_UTIL/bin

E.3 Environment Variables and Files

Each user of VFCS should have the file *Vfcsdefaults* in his or her home directory in the workstation network where VFCS is installed.
Further, each user needs the following in the local environment: (For a PARMACS installation, substitute the environment variable *PARMACS_UTIL* for *VFCS_UTIL* in the description below.)
In the *.cshrc* shell script:

- the *VFCS_UTIL* entry
  - For compilation and execution on the SRM as host processor:
    include the line
    setenv VFCS_UTIL utilsdir
    in the shell script (or *.login*) in the user's home directory on the SRM, where *utilsdir* is the directory containing the utilities from *ipsc_dir/bin* as described above;
  - For compilation and execution on a SUN SPARCstation as host processor: include the line
    setenv VFCS_UTIL utilsdir
    in the user's shell script on the SUN network, where *utilsdir* is the directory containing the utilities from *sun_util/bin*

- the *VFCS* entry in the shell script in the local network where VFCS is installed:
  alias vfcs 'install_dir/vfcs -P -ALL /install_dir'
  where *install_dir* is the directory containing the binary files delivered in the tar file. The argument -P is optional: if it appears, the system will always name the executable files it is using when it is invoked.

In the *.bashrc* shell script:

- the *VFCS_UTIL* entry

- For compilation and execution on the SRM as host processor: include the line
  export VFCS_UTIL="utilsdir"
  in the environment settings (above file or the *.login* file in the user's home directory) on the SRM, where *utilsdir* is the directory containing the utilities from *ipsc_dir/bin* as described above;
- For compilation and execution on a SUN SPARCstation as host processor: include the line
  export VFCS_UTIL="utilsdir"
  in the environment settings on the SUN network, where *utilsdir* is the directory containing the utilities from *sun_util/bin*

- the *VFCS* entry in the shell script in the local network where VFCS is installed:
  alias vfcs='install_dir/vfcs -P -ALL /install_dir'
  where *install_dir* is the directory containing the binary files delivered in the tar file. The argument -P is optional: if it appears, the system will always name the executable files it is using when it is invoked.