VIENNA FORTRAN – A LANGUAGE SPECIFICATION* VERSION 1.1

Hans Zima[a] **Peter Brezany**[a] **Barbara Chapman**[a] **Piyush Mehrotra**[b]
Andreas Schwald[a]

[a]*Department of Statistics and Computer Science,*
University of Vienna, Brünner Strasse 72, A-1210 VIENNA AUSTRIA

[b]*ICASE, MS 132C, NASA Langley Research Center, Hampton VA. 23665 USA*

Abstract

This document presents the syntax and semantics of Vienna Fortran, a machine-independent language extension to FORTRAN 77, which allows the user to write programs for distributed-memory systems using global addresses. Vienna Fortran includes high-level features for specifying virtual processor structures, distributing data across sets of processors, dynamically modifying distributions, and formulating explicitly parallel loops. The language is based upon the Single-Program-Multiple-Data (SPMD) paradigm, which exploits the parallelism inherent in many scientific codes. A substantial subset of the language features has already been implemented.

Keywords: distributed-memory multiprocessor systems, numerical computation, data parallel algorithms, data distribution, alignment, parallel loops, concurrent input/output

*The work described in this paper is being carried out as part of the research project "Virtual Shared Memory for Multiprocessor Systems with Distributed Memory" funded by the Austrian Research Foundation (FWF) under the grant number P7576-TEC and the ESPRIT Project "An Automatic Parallelization System for Genesis", funded by the Austrian Ministry for Science and Research (BMWF). This research was also supported by the National Aeronautics and Space Administration under NASA contract NAS1-18605 while the authors were in residence at ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665. The authors assume all responsibility for the contents of the paper.

Contents

1 Introduction **4**
- 1.1 The Language Features . . . 5
- 1.2 Implementation Status . . . 9
- 1.3 Related Work . . . 10

2 The Model **12**
- 2.1 The Data Space of a Program . . . 12
- 2.2 Processors . . . 13
- 2.3 Distributions . . . 13
- 2.4 Alignment . . . 15

3 Basic Language Specification **16**
- 3.1 Introduction . . . 16
  - 3.1.1 Syntax Metalanguage . . . 16
  - 3.1.2 Basic Elements . . . 17
- 3.2 Processor Declarations . . . 20
  - 3.2.1 Syntax . . . 20
  - 3.2.2 Semantics . . . 20
- 3.3 Processor References . . . 22
  - 3.3.1 Syntax . . . 22
  - 3.3.2 Semantics . . . 22
- 3.4 Distribution Expressions . . . 23
  - 3.4.1 Syntax . . . 23
  - 3.4.2 Overview . . . 24
  - 3.4.3 Basic Intrinsic Distribution Functions . . . 25
  - 3.4.4 Distribution Extraction . . . 27
  - 3.4.5 Distribution Type Definitions . . . 27
  - 3.4.6 Composite Distributions . . . 27
  - 3.4.7 Evaluation of Distribution Expressions . . . 29
- 3.5 Alignment Specifications . . . 30
  - 3.5.1 Syntax . . . 30
  - 3.5.2 Introduction . . . 30
  - 3.5.3 Alignment Expressions . . . 31
  - 3.5.4 Functional Alignment . . . 32
- 3.6 Static Array Annotations . . . 33
  - 3.6.1 Syntax . . . 33
  - 3.6.2 Semantics . . . 33
- 3.7 Dynamically Distributed Arrays . . . 35
  - 3.7.1 Syntax . . . 35
  - 3.7.2 Dynamic Array Annotations . . . 35
  - 3.7.3 Distribute Statements . . . 37
- 3.8 Control Constructs . . . 38
  - 3.8.1 Syntax . . . 38
  - 3.8.2 Introduction . . . 39
  - 3.8.3 The DCASE Construct . . . 39
  - 3.8.4 The IF Construct . . . 42
- 3.9 Allocatable Arrays . . . 43
  - 3.9.1 Syntax . . . 43
  - 3.9.2 Semantics . . . 43

3.10 Procedures . . . 44
3.10.1 Syntax . . . 44
3.10.2 Semantics . . . 44
3.11 Common Blocks . . . 49

4 FORALL Loops **51**
4.1 Syntax . . . 51
4.2 Semantics . . . 51
4.3 Work Distribution . . . 53
4.4 Reduction Operators . . . 54

5 Specification of Distribution and Alignment Functions **56**
5.1 Specification of Distribution Functions . . . 56
5.1.1 Syntax . . . 56
5.1.2 Semantics . . . 56
5.1.3 Examples . . . 58
5.2 Specification of Alignment Functions . . . 61
5.2.1 Syntax . . . 61
5.2.2 Semantics . . . 61

6 Concurrent Input/Output Statements **63**
6.1 Syntax . . . 63
6.2 Semantics . . . 63

A Examples **70**
A.1 Gaussian Elimination . . . 70
A.2 ADI Iteration . . . 73
A.3 Sweep over an Unstructured Mesh . . . 75

B Intrinsic Functions **77**
B.1 ALL . . . 77
B.2 ALLOCATED . . . 77
B.3 ANY . . . 77
B.4 BLOCK . . . 77
B.5 B_BLOCK . . . 77
B.6 CYCLIC . . . 78
B.7 CYCLIC_LEN . . . 78
B.8 DISTRIBUTED . . . 78
B.9 DYNAMIC . . . 78
B.10 IDT . . . 78
B.11 IDTA . . . 79
B.12 INDIRECT . . . 79
B.13 LBOUND . . . 79
B.14 OWNED . . . 79
B.15 OWNER . . . 79
B.16 SIZE . . . 79
B.17 S_BLOCK . . . 80
B.18 UBOUND . . . 80
B.19 $MY_PROC . . . 80
B.20 $NP . . . 80

C Syntax **81**
C.1 Syntax Metalanguage . . . 81
C.2 Basic Elements . . . 81
C.3 Processor Declarations . . . 82
C.4 Processor References . . . 82
C.5 Distribution Expressions . . . 82
C.6 Alignment Specifications . . . 82
C.7 Static Array Annotations . . . 83
C.8 Dynamically Distributed Arrays . . . 83
C.9 Control Constructs . . . 83
C.10 Allocatable Arrays . . . 84
C.11 Procedures . . . 84
C.12 FORALL Loops . . . 84
C.13 Specification of Distribution Functions . . . 85
C.14 Specification of Alignment Functions . . . 85
C.15 Concurrent Input/Output Statements . . . 85

1 Introduction

In recent years, distributed-memory multiprocessing systems have gained an increasing share of the high performance computer market. These architectures are relatively inexpensive to build, and are potentially scalable to very large numbers of processors. They may well prove to be the tools with which the Grand Challenges of Computational Science can be successfully attacked.

The most important single difference between distributed memory systems 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 efficiency. In particular, the management of data, with the twin goals of both spreading the computational workload and minimizing the communication delays, becomes of paramount importance.

A major difficulty with the current generation of these 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 in many aspects. It is tedious, time-consuming and error prone. It 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 ours) have been put forth in recent months for a set of language extensions to achieve this [8, 12, 22, 30], in particular (but not only) for Fortran, and current compiler research is aimed at implementing them.

A notation for specifying data distributions was developed and used in the SUPERB project to parallelize FORTRAN 77 code [42]. Early research on language extensions for distributing data was performed by Mehrotra [25], Callahan and Kennedy [5], Kennedy and Zima [16] and Zima et al. [43]. The issues of data distribution and data movement were analyzed within the framework of the Crystal programming language [9].

Research in compiler technology has so far resulted in the development of a number of prototype systems which are able to convert programs written using global data references to code for distributed memory systems. These include SUPERB [13, 42], Kali [18, 17], and the MIMDizer [27]. These systems require the user to specify the distribution of the program's data. The 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 distributed memory multiprocessor. The compiler analyzes the source code, translating global data references into local and non-local 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 statements and by sending data at the earliest possible point in time.

This document presents the complete syntax and semantics of **Vienna Fortran**, 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 a separate paper [4]; its specification is under development. Since the performance of an SPMD program is profoundly influenced by the distribution of its data, most of the extensions are introduced to permit explicit control of this by the user. A parallel loop is provided, as are means to distribute the work in this loop. 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 and a simple algorithm can be parallelized by the addition of just a few constructs which distribute the program's data across the machines.

In the rest of this section, we give a brief overview of the language elements, in a form which is of necessity incomplete. This should also serve as an introduction to the language for those who do not wish to study

the specification details[1]. We then discuss the implementation of the language features and look at related work. The remainder of the document is devoted to the language extensions, whose syntax and semantics are developed with a few examples.

Section 2 introduces the basic model for data distribution underlying Vienna Fortran. Section 3, which forms the bulk of this document, then introduces the basic language. This includes specifying and referring to processors, the various methods provided for distributing data, a discussion of dynamic distribution, the use of allocatable arrays, the transfer of distributed arrays across procedure boundaries, and handling of common blocks. The discussion of each topic begins with the corresponding syntax, followed by the semantic rules and restrictions, together with some examples of constructs illustrating the syntax and their interpretation. Section 4 which follows introduces an explicitly parallel loop; Section 5 then provides the framework by which a user may construct his or her own distribution and alignment functions. The extensions for reading and writing distributed files are defined in Section 6. Appendix A provides several example codes written in Vienna Fortran, Appendix B lists the intrinsic functions provided for the user in the language extensions, and finally, Appendix C reproduces the entire syntax of the preceding sections.

1.1 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,
  - where a particular data object is stored, or
  - as determined by the compiler.
- Arrays in **common blocks** may be distributed.

[1] The reader is also referred to the examples in Appendix A and in [6, 7] which demonstrate the capabilities of the language features.

- **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.

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)

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.

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**.

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.

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.

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.

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))

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.

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.

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.

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.

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 they 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.

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.

1.2 Implementation Status

The Vienna Fortran Compilation System is currently being developed at the University of Vienna. It is based upon previous work performed by several groups, but, in particular, upon the experience gained with the parallelization system SUPERB ([42]). It currently generates code for the Intel iPSC/860, the GENESIS architecture, and SUPRENUM.

The implementation of a substantial subset of Vienna Fortran has already been completed. This includes

- Static array distributions
- Arbitrary rectilinear block distributions
- Inherited distributions for dummy array arguments
- Forall loops

Special consideration has been given to optimizing the generated code. In particular, the following analysis and optimization methods have been implemented:

- Interprocedural communication analysis
- Communication optimization: matching access patterns to aggregate communication routines, elimination of redundant communication, fusion of communication statements
- Interprocedural dynamic distribution analysis
- Interprocedural distribution propagation
- Procedure Cloning
- Optimization of parallel loop scheduling
- Optimization of irregular access patterns, based on the PARTI routines (cf. [36]).

This compilation system is a full implementation of FORTRAN 77. Among other things, it permits the user to distribute work arrays, sections of which may be individually distributed; it also handles equivalencing. It performs extensive data dependence analysis and interprocedural analysis to determine the correctness of all transformations applied to the program code.

Implementation of further features of Vienna Fortran, in particular the dynamic distributions, is under way. There is still an amount of research to be done in this area, including methods for the efficient handling of user defined distribution and alignment functions.

1.3 Related Work

We discuss some of the related research in both language development for parallel machines and compilation techniques briefly below.

A number of parallel programming languages have been proposed, both for use on specific machines and as general languages supporting some measure of portability (e.g. OCCAM [31]). Languages for coordinating individual threads of a parallel program, such as LINDA [1] and STRAND [11], have been introduced to enable functional parallelism. Most manufacturers have extended sequential languages, such as Fortran and C, with library routines to manage processes and communication. In most explicitly parallel languages, the user performs many of the tasks which a compiler is expected to handle for a Vienna Fortran program.

The concept of defining processor arrays and distributing data to them was first introduced in the programming language BLAZE [19] in the context of shared memory systems with non-uniform access times. This research was continued in the Kali programming language [26] for distributed memory machines, which requires that the user specify data distributions in much the same way that Vienna Fortran does. It permits both standard and user-defined distributions; a forall statement allows explicit user specification of parallel loops. The design of Kali has greatly influenced the development of Vienna Fortran.

Other languages have taken a similar approach: the language DINO [34, 35], for example, requires the user to specify a distribution of data to an *environment*, several of which may be mapped to one processor. The programmer does not specify communication explicitly, but must mark non-local accesses. In Booster [28, 29], data distributions are specified separately from the algorithm in an *annotation module*; a distinction is made between work and data partitions.

More recently, the Yale Extensions, currently being developed by Chen et al. [8], specify the distribution of arrays in three stages: alignment, partition and a physical map. Because all these stages are modeled as bijective functions between index domains, data replication is not possible. By restricting the scope of layout directives to phases, a block structure is imposed on Fortran 90.

The programming language Fortran D [12], under development at Rice University, proposes a Fortran language extension in which the programmer specifies the distribution of data by aligning each array to a virtual array, known as a decomposition, and then specifying a distribution of the decomposition to a virtual machine. These are executable statements, and array distributions are dynamic only. While the general use of alignment enables simple specification of some of the relationships between items of program data, we believe that it is often simpler and more natural to specify a direct mapping. We further believe that many problems will require more complete control over the way in which data elements are mapped to processors at run time. Fortran90D [41], proposed by researchers at Syracuse University, is based upon CM Fortran [38].

Digital Equipment Corporation has proposed language extensions [22] for data distribution conformant with both FORTRAN 77 and Fortran 90. These include directives for statically aligning data with decompositions. They are specified when the array is declared. The user may explicitly distribute dummy array arguments; if the distribution differs from that of the actual argument, redistribution occurs. The original distribution is restored at subroutine exit. It is assumed that the compiler will implement a default distribution for those arrays which are not explicitly distributed by the user. A forall statement is provided.

Cray Research Inc. has announced a set of language extensions to Cray Fortran (cf77) [30] which enable the user to specify the distribution of data and work. They provide intrinsics for data distribution and permit redistribution at subroutine bounds. Further, they permit the user to structure the executing processors by giving them a shape and weighting the dimensions. Several methods for distributing iterations of loops are provided.

The Cray programming model assumes that initial execution is sequential and the user specifies the start and end of parallel execution explicitly. Many of the features of shared memory parallel languages have been retained: these include critical sections, events and locks. New instructions for node I/O are provided. In addition, there are a number of intrinsic functions to access parts of arrays local to a processor, and reduction and parallel prefix operations are included.

The implementation of Vienna Fortran and similar languages requires a particularly sophisticated compilation system, which not only performs standard program analysis but also, in particular, analyzes the

program's data dependences [44]. In general, a number of code transformations must be performed if the target code is to be efficient. The compiler must, in particular, insert all messages - optimizing their size and their position wherever possible.

The compilation system SUPERB (University of Vienna) [42] takes, in addition to a sequential Fortran program, a specification of the desired data distribution and converts the code to an equivalent program to run on a distributed memory machine, inserting the communication required and optimizing it where possible. The user is able to specify arbitrary block distributions. It can handle much of the functionality of Vienna Fortran with respect to static arrays.

The Kali compiler [18] was the first system to support both regular and irregular computations, using an inspector/executor strategy to handle indirectly distributed data. It produces code which is independent of the number of processors.

The MIMDizer [27] and ASPAR [15] (within the Express system) are two commercial systems which support the task of generating parallel code. The MIMDizer incorporates a good deal of program analysis, and permits the user to interactively select block and cyclic distributions for array dimensions. ASPAR performs relatively little analysis, and instead employs pattern-matching techniques to detect common stencils in the code, from which communications are generated.

Pandore [2] takes a C program annotated with a user-declared virtual machine and data distributions to produce code containing explicit communication. Compilers for several functional languages annotated with data distributions (Id Nouveau [33], Crystal [21]) have also been developed which are targeted to distributed memory machines.

Quinn and Hatcher [14], and Reeves et al. [10, 32] compile languages based on SIMD semantics. These attempt to minimize the interprocessor synchronizations inherent in SIMD execution. The AL compiler [39], targeted to one-dimensional systolic arrays, distributes only one dimension of the arrays. Based on the one dimensional distribution, this compiler allocates the iterations to the cells of the systolic array in a way that minimizes inter-cell communications.

The PARTI primitives, a set of run time library routines to handle irregular computations, have been developed by Saltz and coworkers [36, 37]. These primitives have been integrated into a compiler and are also being implemented in the context of the FORTRAN D Programming environment being developed at Rice University. Similar strategies to preprocess DO loops at runtime to extract the communication pattern have also been developed within the context of the Kali language by Koelbel and Mehrotra [18, 20]. Explicit run-time generation of messages is also considered by [10, 21, 33], however, these do not save the extracted communication pattern to avoid recalculation.

2 The Model

2.1 The Data Space of a Program

Definition 1 *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 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 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 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.*[2]
   (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.*

[2] 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 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.* □

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 (see 3.1.2).

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 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".

Processor declarations and processor references will be discussed in Sections 3.2 and 3.3, respectively.

2.3 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 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 7 Distributions

1. *Let $A \in \mathcal{A}$ denote an* **allocated** *array, and assume that R is a processor array. An index mapping δ_R^A 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. For most of this paper 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 8 *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 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 (see Section 3.6)).

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.

In much of our treatment of distributions in later sections of this document, we will deal with *classes of distributions* rather than individual distributions. These classes will be called **distribution types**; they are the result of certain abstractions which are related to the way distributions are expressed in the language (see Section 3).

A distribution for an array determines for each processor which of the array elements are local to the processor, and which are not. While access to local data items can be performed via normal memory references, nonlocal objects can be read or written only via message passing, which, for most of today's architectures, is a highly expensive operation. Thus, while the selection of a distribution does not affect the correctness of a program, it plays a pivotal role for its run-time performance. The amount of communication can be reduced if data objects are replicated, i.e., if they can be owned by two or more processors. This is the reason why in our model distributions map an array index domain into the **powerset** of the processor index domain, rather than simply into the processor index domain.

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.

2.4 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 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 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 $\mathbf{i} \in \mathbf{I}^A$:

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

□

3 Basic Language Specification

3.1 Introduction

Section 3 defines the set of Fortran extensions that constitute the basic Vienna Fortran language. They include:

- Assertions (Section 3.1.2)
- Array sections (Section 3.1.2)
- Processor declarations and references (Sections 3.2 and 3.3)
- Static and dynamic array annotations (Sections 3.4, 3.5 and 3.6)
- Dynamic distributions of arrays and their control (Section 3.7)
- Allocatable arrays and the allocate and deallocate statements (Section 3.9)
- Dummy array annotations and the mechanisms for transferring distributed arrays to procedures (Section 3.10)
- Common blocks (Section 3.11)

These topics will be discussed in individual subsections below.

In the remainder of this introduction we will specify the syntax metalanguage, describe a number of basic concepts that will be used throughout the document, and give an informal introduction into the declaration annotation syntax.

3.1.1 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 ([3]).[3]
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

[3] In some cases, the meaning may be extended to include constructs of the extension.

3.1.2 Basic Elements

Syntax

1. assertion → **ASSERT** "(" *expression* ")"
2. array-section → *array_name* ["("section-subscript-list")"]
3. section-subscript → subscript | subscript-triplet
4. subscript → *integer_expr*
5. subscript-triplet → [subscript] ":" [subscript] [":" stride]
6. stride → *integer_expr*
7. data-reference → *array_element_name* | array-section
8. generalized-array-declarator → *array_declarator* | assumed-shape-array-declarator
9. assumed-shape-array-declarator → *array_name* "(" assumed-shape-spec-list ")"
10. assumed-shape-spec → [*dim_bound_expr*] ":"
11. declaration-annotation → actual-array-annotation | dummy-array-annotation
12. actual-array-annotation → static-array-annotation | dynamic-array-annotation
13. extension-executable-statement → distribute-statement | allocate-statement | deallocate-statement | forall-loop | dcase-construct | concurrent-io-statement

This section describes a number of loosely related basic concepts of the language, which will be used throughout the document.

Assertions

An *assertion* is specified in the form:

ASSERT (*expr*)

where *expr* is an logical expression. An assertion can occur in any place of the program where a statement is allowed. It specifies a relationship between objects of the program which the compiler can use to improve code generation. An assertion that specifies a relationship which is not universally **true** may result in a compile-time or run-time error.

Array Sections

Array sections are allowed as actual arguments in procedure calls, and can also be used to specify sets of processors (see Section 3.3).
An *array section* represents an array.
If an array section consists only of an *array_name*, then it represents the whole array associated with this name.
Otherwise, it has the form:

$$A(ss_1, \ldots, ss_n)$$

where A is an array name representing an array of rank n, and each ss_j is a *section-subscript*. The number of *subscript-triplets*, m, in the array section must be at least 1. Let AS denote the array represented by the array section. Then AS has rank m. It consists of all elements of A determined by all possible subscript lists obtainable from the single subscripts or sequences of subscripts specified by each section subscript. This set must be nonempty.

Let $d_1, \ldots, d_m$ denote the positions in which a subscript triplet occurs. Then $\mathbf{I}^{AS}_{d_j} \subseteq \mathbf{I}^{A}_{d_j}$ for all $j, 1 \leq j \leq m$. We explain now how the index domain of AS is constructed by specifying the sequence of subscript values determined by a subscript triplet.

Consider a subscript triplet occurring in position d_j of the section subscript list, and assume that $\mathbf{I}^{A}_{d_j} = [L : U]$. A subscript triplet has the form:

$$[sub_1] : [sub_2][: str]$$

The default values are: $sub_1 = L$, $sub_2 = U$ and $str = 1$. The stride str must not be 0.
The subscript triplet determines a sequence of values in the following way:

- If $str > 0$, then $sub_1 \leq sub_2$ must hold. The sequence begins with sub_1 and proceeds in increments of str to the largest such integer not greater than sub_2.
- If $str < 0$, then $sub_2 \leq sub_1$ must hold. The sequence begins with sub_2 and proceeds in increments of str to the smallest such integer not less than sub_1.

Note that $\mathbf{I}^{AS}$ is not necessarily a standard index domain, and that the order of elements within a dimension of A may be reversed in AS. Furthermore, sub_1 and sub_2 need not be within the declared bounds of A if all values used in selecting array elements are within the declared bounds.

Example 1 Array sections
Consider the declarations of $A1$ and $A2$, as given below:

REAL $A1(100),\ A2(100, 100)$

The following are examples of array sections:

- $A1$ — *this designates the whole array $A1$*
- $A1(40 : 60)$ — *this specifies the one-dimensional subarray with elements $A1(40), A1(41), \ldots, A1(60)$.*
- $A1(90 : 102 : 5)$ — *this specifies the one-dimensional subarray containing the elements $A1(90), A1(95), A1(100)$.*
- $A1(100 : 1 : -1)$ — *this is $A1$, with its elements in reverse order.*
- $A1(50 : 50)$ — *this is the one-dimensional subarray containing the single element $A1(50)$.*
- $A2(1 : 10, 91 : 100)$ — *a two-dimensional section of $A2$, containing the elements $A2(1, 91), A2(2, 91), \ldots, A2(10, 100)$.*
- $A2(I, :)$ — *the I-th row of $A2$: this is a one-dimensional section.*
- $A2(:, J)$ — *the J-th column of $A2$, a one-dimensional section.*
- $A2(:, :)$ — *equivalent to the whole array, $A2$.*

□

Data References

A *data reference* is either an *array_element_name* or an *array-section*. The concept will be used in the context of data arrays and processor arrays.

Generalized Array Declarators

Generalized array declarators are either FORTRAN 77 *array_declarator*s or *assumed-shape-array-declarator*s. They are used to specify processor arrays (Sections 3.2 and 3.10) and dummy arguments of user-specified distribution and alignment functions (see Sections 5.1 and 5.2).

For *array_declarator*s, standard FORTRAN 77 rules apply.

Consider now an *assumed-shape-array-declarator*:

$$A(spec_1, \ldots, spec_n)$$

Here, A is an array name, and each $spec_i, 1 \leq i \leq n$, is an *assumed-shape-spec* of the form

$$[lwb_i] :$$

where lwb_i is a *dim_bound_expr*. The context of an assumed shape array declarator always specifies a uniquely determined **actual argument array**, B, of the same rank as A. The shape of A is defined so as to be equal to the shape of B.

Let i be arbitrarily selected, and assume that the bounds of the actual argument array B in dimension i are given as $[l : u]$, and define the bounds to be associated with dimension i of A as $l' : u'$. The meaning of an assumed shape spec is then defined as follows:

1. If the assumed shape spec is of the form lwb_i :, and c is the value of lwb_i, then $l' := c$ and $u' := u+c-l$.
2. If the assumed shape spec is of the form ":", then $l' := l$ and $u' := u$.

The bounds and extents associated with the index domain of A can be determined at run time by referencing the intrinsic functions *LBOUND, UBOUND*, and *SIZE* (see Appendix B).

Example 2 *Assume an actual argument array* $B(5 : 10, 11 : 20)$. *The following constructs are legal assumed shape array declarators, and have a meaning as explained below:*

- $A(:,:)$ *specifies an array with bounds* $A(5 : 10, 11 : 20)$
- $A(1 :,:)$ *specifies the bounds of* A *as* $A(1 : 6, 11 : 20)$

The following constructs are illegal:

- $A(N,:)$
- $A(:)$ *(the rank of* A *must be* 2*).* □

Declaration Annotations

Vienna Fortran allows annotations to be appended to array declarations. Such annotations specify the method by which the arrays introduced in the declaration are to be distributed.

Declaration annotations include *actual* and *dummy* array annotations.

Actual array annotations are either **static** or **dynamic**. They characterize arrays as statically or dynamically distributed, and define attributes of their distributions. Dummy array annotations are associated with dummy arguments of procedures.

The details of the syntax and semantics of declaration annotations will be defined in later parts of this section. Here, we informally (and incompletely) outline some characteristics of static annotations to provide the reader with some intuition and help him to understand the examples.

Consider a FORTRAN 77 array declaration of the form[4]:

$$\textbf{REAL } ad_1, ad_2, \ldots, ad_r$$

where the $ad_i, 1 \leq i \leq r$, are *array_declarators*, specifying array identifiers and their associated index domains. A **static array annotation**, appended to such a declaration, applies to all these arrays; as a consequence, we need to consider only annotated declarations with exactly one array A. Such an annotation may have the form

$$\textbf{REAL } A(l_1 : u_1, \ldots, l_n : u_n) \textbf{ DIST } (dex) \textbf{ TO } pref$$

where *dex* is a **distribution expression** and *pref* a **processor reference**. Any or all of the components of an annotation may be omitted, but we defer the discussion of defaults to a later section (see Section 3.6). The distribution expression *dex* specifies a **distribution type**, which, together with *pref* and $\mathbf{I}^A$ determines a distribution for A. For example, in the specification:

REAL B(N) **DIST** ($BLOCK$) **TO** R

the annotation contains the distribution expression ($BLOCK$) and a reference to a processor array R. This specifies that B is to be partitioned into evenly-sized contiguous blocks, which are to be distributed to all of the processors in R.

The next two subsections will discuss processor declarations (Section 3.2) and processor references (Section 3.3); after that we will discuss various methods for specifying a distribution (Section 3.4 and 3.5).

3.2 Processor Declarations

3.2.1 Syntax

1. processor-declaration → primary-processor-structure [secondary-processor-structures]
2. primary-processor-structure → **PROCESSORS** generalized-array-declarator
3. secondary-processor-structures → **RESHAPE** generalized-array-declarator–list

3.2.2 Semantics

Each program unit of a Vienna Fortran program may contain a *processor-declaration*, thereby introducing one or more **processor arrays**. Processor arrays serve the following purposes:

1. To define the set of processors, P, on which the program will execute.
2. To impose one or more virtual processor structures on P (in the form of multi-dimensional arrays), thus providing different views of the processor set.
3. To provide a means for accessing P and its elements in the program.

The form of a processor declaration is:

[4] REAL arrays are generally used to illustrate the syntax and semantics of annotations. This can be immediately generalized to the other variants of array declarations in FORTRAN 77.

PROCESSORS gd_1 [**RESHAPE** $gd_2, \ldots, gd_r$]

where the $gd_i, 1 \leq i \leq r$ are *generalized-array-declarator*s (see Section 3.1.2) that satisfy the following rules:

1. *Assumed size array declarator*s are not allowed.
2. *Dimension bound expressions* may contain references to the intrinsic function $\$NP$. They may also contain references to variable names as in adjustable array declarators. Any variable-name occurring in such an expression is implicitly declared of type integer. Its scope is the program unit in which the processor declaration occurs. If the processor declaration occurs in the main program, the values of such variables are obtained from the environment. For processor declarations occurring in other program units, the variables used in the declaration must appear in the argument list or in a common block in that subprogram. Their values must not be redefined during program execution.
3. Each gd_i introduces a **processor array**. The processor array associated with gd_1 is called the **primary array**, all other arrays (if any) are called **secondary**.
4. All processor arrays are associated with the same set of processors, P. As a consequence, all processor arrays have the same size and the values assigned to variables used in the dimension bound expression must not violate this condition. Furthermore, for any two processor arrays, the array element ordering (column major order), as specified in Fortran, determines a one-to-one correspondence between their elements.

For any processor array R occurring in the processor declaration, the evaluation of the array declarators yields an associated standard index domain $\mathbf{I}^R$.

There is always an implicit one-dimensional processor array declaration of the form $\$P(1 : \$NP)$. If the program does not contain an explicit processor declaration, $\$P$ is understood as the primary array, and no secondary array exists. Otherwise, $\$P$ is a secondary array. A reference to the intrinsic function $\$MY_PROC$ yields the index of the executing processor in $\$P$.

The user or the environment of the program must provide the following information upon program start:

1. If the main program does not contain a processor declaration: the number of processors on which the program is to execute must be specified. A reference to the intrinsic function $\$NP$ yields this value.
2. If the main program does contain a processor declaration:
   (a) For any processor array specified by an assumed shape array declarator of rank n, an array index set of rank n has to be specified.
   (b) Each variable occurring in an adjustable array declarator must be defined.

Note that processor arrays do not imply a specific topology of the actual hardware structure: in particular, they **do not** specify that the processors are physically connected as a mesh.

Assertions may play an important role in processor declarations by supplying the compiler with information on the range of available processors and the values of variables occurring in dimension bound expressions.

Example 3 A two-dimensional primary process structure

```
ASSERT ( NP1 .GE. 8)
PROCESSORS R2(1:NP1, 1:NP1)
```

This declares a two-dimensional processor array R2 with $\mathrm{NP1}^2$ *processors. The value of* NP1 *is asserted to be greater than or equal to* 8*. This constraint has to be satisfied when the value of* NP1 *is defined at the start of program execution.*

Example 4 Processor Reshaping

```
ASSERT ( NP1 .GE. 8)
PROCESSORS R2(1:NP1, 1:NP1) RESHAPE R1(1:NP1*NP1)
```

$R1$ provides a one-dimensional secondary process structure for $R2$. For all i, j with $1 \leq i, j \leq$ NP1, $R2(i, j)$ designates the same processor as $R1(i + (j - 1) * \text{NP1})$.

Note that the explicit reshaping to $R1$, as shown in this example, is actually redundant, since $\$P$ is implicitly declared identically to $R1$. □

The processor declaration of this example will be used consistently throughout the examples of this section, i.e., $R1$ and $R2$ will be used with the meaning defined here.

3.3 Processor References

3.3.1 Syntax

1. processor-reference → processor-element-name | processor-section ["(/"dimension-permutation"/)"]
2. processor-element-name → *array_element-name*
3. processor-section → array-section
4. dimension-permutation → *int_constant_expr*–list

3.3.2 Semantics

Processor references are used in contexts which require that either individual processors or sets of processors are explicitly specified. This includes the **TO**- clause in declaration annotations (Section 3.1.2), the distribute statement (Section 3.7.3), and the **ON**- clause in **FORALL**-loops (Section 4). Processor references may be associated with an arbitrary processor array known at the place where the reference occurs.

A *processor-reference* designates either a *processor-element-name* or a *processor-section*, possibly supplemented by a *dimension-permutation*.

A processor element name designates a single processor by applying a subscript list to a processor array name. This is done in the same way as the specification of an array element in FORTRAN 77.

A processor section is an array section associated with a processor array, followed by an optional dimension permutation. If a dimension permutation is not specified, then the processor section defines a subarray as explained in Section 3.1.2.

A processor specification containing a dimension permutation has the form

$$PS[(/d_1, \ldots, d_m/)]$$

where PS is a processor section of rank m, and $(d_1, \ldots, d_m)$, the dimension permutation, specifies a permutation of $(1, \ldots, m)$. The evaluation of this construct yields a processor array PS' of dimension m such that for all $j, 1 \leq j \leq m$, dimension j of PS' corresponds to dimension d_j of PS.

Example 5 *Based on Example 4, we form the following processor references:*

- *$R2$ — this defines the whole processor array*
- *$R2(/2, 1/)$ — this represents the transposed array $R2$, i.e., the processor in the j-th row and i-th column of $R2(/2, 1/)$ is the same as $R2(i, j)$.*

- $R1(::2)$ — *this represents the section of the one-dimensional processor array containing every other processor, i.e.* $R1(1), R1(3), \ldots$.
- $\$P(1:\$\mathrm{NP}:2)$ — *the same section as above*
- $R2(\mathrm{NP}1, 3)$ — *a processor element name*
- $R1(3*\mathrm{NP}1)$ — *the same processor element as above*
- $R2(2:\mathrm{NP}1-2:2, \mathrm{NP}1-2:0:-4)(/2,1/)$ — *this is a two-dimensional processor section. Assume* $\mathrm{NP}1 = 8$. *Then* $R2(2:\mathrm{NP}1-2:2, \mathrm{NP}1-2:0:-4)$ *specifies the processors*

$$R2(2,6), R2(4,6), R2(6,6), R2(2,2), R2(4,2), R2(6,2)$$

in this order. $R2(2:\mathrm{NP}1-2:2, \mathrm{NP}1-2:0:-4)(/2,1/)$ *transposes the two dimensions: i.e., we obtain an array, say* $R2'$, *with two elements in the first dimension, and three elements in the second dimension. The elements of* $R2'$, *in array element order, correspond to the processors*

$$R2(2,6), R2(2,2), R2(4,6), R2(4,2), R2(6,6), R2(6,2)$$

□

3.4 Distribution Expressions

3.4.1 Syntax

1. distribution-expression → simple-distribution-expression | composite-distribution-expression
2. simple-distribution-expression → distribution-function-reference | distribution-extraction | distribution-type-name
3. distribution-function-reference → *function_reference*
4. distribution-extraction → "=" array-or-dimension
5. array-or-dimension → *array_name* | array-dimension
6. array-dimension → *array_name* dimension-qualifier
7. dimension-qualifier → "." *int_constant_expr*
8. distribution-type-definition → **DTYPE** "(" dtype-pair-list ")"
9. dtype-pair → name "=" "("distribution-expression")"
10. composite-distribution-expression → dimensional-expression-list
11. dimensional-expression → simple-distribution-expression | ":"

3.4.2 Overview

A **distribution expression** specifies a class of distributions which is called a **distribution type**. The application of a distribution type to an environment which specifies a (data) array and a processor reference yields a distribution (see Sections 3.1.2, 3.6, 3.7.3, 5.1).

For example, (*BLOCK*, *CYCLIC*) is a composite distribution expression with two elements, which are references to the intrinsic distribution functions *BLOCK* and *CYCLIC*. The associated distribution type can be applied to an environment that provides a two-dimensional array and a two- or more-dimensional processor section; this yields a distribution. More specifically, in

REAL A(N,M),B(M,N)
DIST (*BLOCK*, *CYCLIC*) **TO** R2

the distribution type associated with $(BLOCK, CYCLIC)$ is applied first to array A and processor array $R2$, and then to array B and $R2$. In both cases, the first dimension of the array is distributed blockwise, and the second dimension cyclically. If N and M are different, the resulting distributions δ^A and δ^B will be different.

Distribution expressions are either **simple** or **composite**. **Simple distribution expressions** fall into three categories:

- **Distribution function references**: A distribution function is a special function used to specify distribution types. The language provides a set of **intrinsic distribution functions**; in addition, the user may specify arbitrary distribution types (see Section 5.1). We will first introduce the **basic intrinsic distribution functions** *BLOCK*, *CYCLIC*, and *INDIRECT*. *BLOCK* and *CYCLIC* specify mappings between one array dimension and one processor array dimension (Section 3.4.3); such functions are called **1-1 distribution functions**. More generally, an **n-m distribution function** maps an n-dimensional array index domain to an m-dimensional processor section, where n and m are both greater than or equal to 1. The types associated with such a function are called n-m distribution types.

  Distribution functions may have parameters, but must be side-effect free; their references follow the normal FORTRAN 77 rules, except that an empty argument list need not be enclosed by parentheses.

- **Distribution extractions** specify a distribution type by referring to the distribution associated with another array, or to that of a dimension of such an array. They will be discussed in Section 3.4.4.

- Finally, distribution types can be bound to a name (see Section 3.4.5); each applied occurrence of such a **distribution type name** stands for the associated distribution type.

Composite distribution expressions are lists of **dimensional expressions**. They specify **composite distribution types**, which are associated with multi-dimensional arrays in such a way that different dimensions are distributed independently. Each entry in a dimensional expression list is either a distribution expression that specifies a mapping from one array dimension to one processor array dimension, or the **elision symbol** ":" that hides one array dimension from distribution. Such an array dimension is referred to as **hidden**.

If a distribution expression is evaluated in a certain environment, all its components must be **defined** in that environment. More specifically:

- in a distribution function reference, the function must be defined and all actual arguments must have a defined value,

- the array occurring in a distribution extraction must be **allocated** and associated with a well-defined distribution,

- a distribution type identifier must be bound to a distribution type,

- all simple distribution expressions occurring in a composite distribution expression must be defined.

3.4.3 Basic Intrinsic Distribution Functions

The **basic intrinsic distribution functions** are *BLOCK*, *CYCLIC*, and *INDIRECT*.

We first discuss *BLOCK* and *CYCLIC*. A reference to one of these functions defines a 1-1 distribution type which maps exactly one array dimension to exactly one processor array dimension. We model the effect of these functions in an environment that provides an array index domain $\mathbf{I}^A = [1:N]$ and a processor array $R1$ with index domain $\mathbf{I}^{R1}$ =[*1:NP1*]. We will define the functions by specifying the associated distributions, which will be simply denoted by δ. In all examples, we will assume *NP1*= 4.

1. **Block Distributions**
   The block distribution function is *BLOCK*. It divides the array into contiguous blocks, whose sizes differ by at most 1. More precisely, let $q := \lfloor \frac{N}{NP1} \rfloor$. If $q = N/NP1$, then the array is divided into *NP1* blocks of size q. Otherwise, we have

   $$N = q.NP1 + r = (q+1)r + q(NP1 - r), \text{ where } 0 < r < NP1.$$

   In this case, the array is divided into r blocks of size $q+1$ — which are mapped to the first r processors — and $NP1 - r$ blocks of size q, mapped to the remaining processors. The distribution function thus can be defined as follows:

   (a) **Case 1:** $q = N/NP1$: $\delta(i) = \{\lceil \frac{i}{q} \rceil\}$ for all $i, 1 \leq i \leq N$

   (b) **Case 2:** $q < N/NP1$. Let $N' := (q+1)r$:
   - $\delta(i) = \{\lceil \frac{i}{q+1} \rceil\}$ for all i such that $1 \leq i \leq N'$
   - $\delta(i) = \{\lceil \frac{i-N'}{q} + r \rceil\}$ for all i such that $N' < i \leq N$.

   Example 6 : Block Distributions

   REAL A(12) **DIST** (*BLOCK*) **TO** R1
   REAL B(13) **DIST** (*BLOCK*) **TO** R1

   We assume NP1 = 4. *For A, we have* $q = 3$ *and Case 1:*

   - $\delta^A(i) = \{1\}$ *for* $1 \leq i \leq 3$
   - $\delta^A(i) = \{2\}$ *for* $4 \leq i \leq 6$
   - $\delta^A(i) = \{3\}$ *for* $7 \leq i \leq 9$
   - $\delta^A(i) = \{4\}$ *for* $10 \leq i \leq 12$

   For the second array, we obtain $q = 3$ *and Case 2, with* $r = 1$, *applies:*

   - $\delta^B(i) = \{1\}$ *for* $1 \leq i \leq 4$
   - $\delta^B(i) = \{2\}$ *for* $5 \leq i \leq 7$
   - $\delta^B(i) = \{3\}$ *for* $8 \leq i \leq 10$
   - $\delta^B(i) = \{4\}$ *for* $11 \leq i \leq 13$.

   □

   For an explicit specification of the function *BLOCK* see Section 5.1.3. Additional intrinsic functions provide the means for specifying arbitrary rectilinear distributions, including irregular ones (see Section B).

2. **Block-Cyclic and Cyclic Distributions**

Block-cyclic distributions are specified by the function *CYCLIC*, with an argument, $l \geq 1$, of type integer. *CYCLIC(l)* defines contiguous segments of length l and maps them cyclically to the processors. The distribution function is given as follows:

$$\delta(i) = \{MODULO(\lceil \frac{i-1}{l} \rceil, NP1 + 1)\} \text{ for all } i, 1 \leq i \leq N$$

Example 7 Block-Cyclic Distribution

REAL D(13) **DIST** (*CYCLIC*(2)) **TO** R1

- $\delta^C(i) = \{1\}\ for\ i = 1, 2, 9, 10$
- $\delta^C(i) = \{2\}\ for\ i = 3, 4, 11, 12$
- $\delta^C(i) = \{3\}\ for\ i = 5, 6, 13$
- $\delta^C(i) = \{4\}\ for\ i = 7, 8$

□

References to *CYCLIC* may be written without specifying an argument: in this case, an argument value of $l = 1$ is assumed, and we speak of a **cyclic** distribution.

Example 8 Cyclic Distribution

REAL C(12) **DIST** (*CYCLIC*) **TO** R1

- $\delta^C(i) = \{1\}\ for\ i = 1, 5, 9$
- $\delta^C(i) = \{2\}\ for\ i = 2, 6, 10$
- $\delta^C(i) = \{3\}\ for\ i = 3, 7, 11$
- $\delta^C(i) = \{4\}\ for\ i = 4, 8, 12$ □

□

3. **Indirect Distributions**

Some large and computationally intensive problems are characterized by dynamically varying data structures and/or irregular access patterns.

One method that can be applied in such a situation is the representation of the distribution of an array by means of a **mapping array**, C, which is defined and used at run-time. We call this an **indirect distribution** and model it via references to the intrinsic distribution function *INDIRECT*.

Assume that *INDIRECT(C)* is to be applied to an array A. C, the mapping array, is an integer array that must satisfy $\mathbf{I}^A \subseteq \mathbf{I}^C$. It is understood to define a mapping from $\mathbf{I}^A$ to the implicit linear processor array $\$P(1 : \$NP)$ by associating with every index in $\mathbf{I}^A$ the corresponding value of C. More precisely, for all $\mathbf{i} \in \mathbf{I}^A$, $\delta^A_{\$P}(\mathbf{i}) = \{C(\mathbf{i})\}$.

An example for the application of this function is given in Section 3.7.3.

3.4.4 Distribution Extraction

A **distribution extraction** is specified in the form

$$= A[.dim]$$

where A is an array followed by an optional dimension specifier. At the time of evaluation of this construct, A must be associated with a well-defined distribution, say δ^A. If dim is specified, δ^A must be composite, and dim an integer constant expression with a value between 1 and $rank(A)$.

If the dimension specifier is not present, then the distribution extraction yields the distribution type associated with δ^A. Otherwise, it yields the distribution type of the dimension of A determined by the value of dim.

Example 9 Distribution Extraction

REAL B(N) **DIST** (*BLOCK*) **TO** R2
REAL E(2*N) **DIST** (=B)

The second declaration is equivalent to:

REAL E(2*N) **DIST** (*BLOCK*) **TO** R2

i.e., both B and E will be distributed by BLOCK *. Note that the block sizes for the arrays B and E will be different since the extents of the two arrays are different.* □

3.4.5 Distribution Type Definitions

A **distribution type definition** has the form

$$\textbf{DTYPE}\ (t_1 = (dex_1), \ldots, t_n = (dex_n))$$

where the t_i and $dex_i, 1 \leq i \leq n$, respectively denote names and distribution expressions. For each i, dex_i is evaluated, and the resulting distribution type is bound to t_i. Any arguments for distribution functions occurring in dex_i are evaluated at this time. The t_i are called **distribution type names**; every reference to such a name represents the associated distribution type.

Example 10 Distribution Type Definition

DTYPE(BLK3= (*BLOCK,BLOCK,BLOCK*), REPS= (*CYCLIC(K),:,CYCLIC(2*K),:*))

*introduces identifiers BL3 and REPS, which represent the distribution types associated with the distribution expressions (*BLOCK,BLOCK,BLOCK *) and (*CYCLIC(K'),:,CYCLIC(2*K'),:*), respectively, where K' denotes the value of K at the time the distribution type definition is evaluated.* □

3.4.6 Composite Distributions

In this section we will define the concept of a **composite distribution**. Such a distribution can be represented as a tuple of independent mappings between single array and single processor array dimensions. In the subsequent section, we will see how composite distribution expressions are evaluated to determine classes of such distributions, which will be called composite distribution types.

We begin with an example illustrating some of the basic rules underlying composite distributions.

Example 11 Composite Distributions

REAL B(N,N) **DIST** (*BLOCK, CYCLIC*) **TO** R2

Dimension i, $i = 1, 2$, of array B is mapped to dimension i of $R2$; the first dimension is to be distributed by $BLOCK$, and the second cyclically. If NP1 *divides N and $q = N/\mathrm{NP1}$, then this specifies a distribution δ such that*

$$\delta^B(i,j) = \{\lceil \tfrac{i}{q} \rceil, MODULO(j-1, \mathrm{NP1}) + 1\}$$

for all i, j with $1 \leq i, j \leq N$.

In

REAL C(N,N,N) **DIST** (*BLOCK, CYCLIC, :*) **TO** R2

the third dimension of C is hidden – which is indicated by the elision symbol ":"–, whereas the first and second dimensions of C are mapped to the first and second dimensions of $R2$ in the same way as for B above. Thus we obtain the distribution function

$$\delta^C(i,j,k) = \{\lceil \tfrac{i}{q} \rceil, MODULO(j-1, \mathrm{NP1}) + 1\}$$

for all i, j, k such that $1 \leq i, j, k \leq N$.

With another example, we illustrate **replication** *across a processor dimension. Consider*

REAL D(N) **DIST** (*BLOCK*) **TO** R2

In this case, the first dimension of D is mapped to the first dimension of $R2$, and replication is performed across all processors in the second dimension of $R2$. This yields:

$$\delta^D_{R2}(i) = \{(\lceil \tfrac{i}{q} \rceil, r_2) \mid 1 \leq r_2 \leq \mathrm{NP1}\} \text{ for all } i \ (1 \leq i \leq N)$$

□

We now define composite distributions precisely:

Definition 12 *Let*

1. *$A \in \mathcal{A}$ denote an array of rank n with index domain $\mathbf{I}^A = [l_1 : u_1, \ldots, l_n : u_n]$,*
2. *R a processor section of rank m with index domain $\mathbf{I}^R = [1 : \mathrm{NP}_1, \ldots, 1 : \mathrm{NP}_m]$, and*
3. *δ a distribution for A with respect to R.*

δ is called a **composite distribution of rank** *n' iff the following conditions are all met:*

1. *Each dimension of A is characterized with respect to δ as either* **distributed** *or* **hidden**. *Let $(d_1, \ldots, d_{n'})$ denote the sequence of* **distributed dimensions**, *where $1 \leq n' \leq n$, and $h := n - n'$ designates the number of* **hidden dimensions**.
2. *$m \geq n'$*
3. *The distributed array dimensions $d_1, \ldots, d_{n'}$ are mapped to the processor dimensions $1, \ldots, n'$ in this order: For each $j, 1 \leq j \leq n'$, there exists a distribution δ_{d_j} from $\mathbf{I}^A_{d_j}$ to $\mathbf{I}^R_j$.*

4. *For each* $\mathbf{i} = (i_1, \ldots, i_n) \in \mathbf{I}^A$*:*

$$\delta(\mathbf{i}) = \{(r_1, \ldots, r_m) \in \mathbf{I}^R \mid \forall j, 1 \leq j \leq n' : r_j \in \delta_{d_j}(i_{d_j})\}$$

Note that for $m > n'$ *the* $m - n'$ *processor dimensions* $n' + 1, \ldots, m$ *are universally quantified.*

5. δ *is represented as an n-tuple, where each position is either an elision symbol or a distribution* δ_{d_j}*, as defined above.*

A distribution which is not composite is called **simple**. □

There are two important special cases for composite distributions, which are characterized by $h = 0$ and $h = n$.

In the first case, $h = 0$, all dimensions of A are distributed; we have $n' = n$, and the sequence $(d_1, \ldots, d_{n'})$ of distributed dimensions is equal to $(1, \ldots, n)$.

In the second case, $h = n$, which is denoted by the distribution expression $(:, \ldots, :)$, consisting of n elision symbols — we have $n' = 0$ and the sequence of distributed dimensions is empty. In this case, the expression for the distribution reduces to

$$\delta(\mathbf{i}) = \mathbf{I}^R,$$

expressing the **total replication** of the array.

3.4.7 Evaluation of Distribution Expressions

Let *dex* denote a distribution expression: then we first specify the evaluation of *dex*, yielding the **distribution type** associated with *dex*; this will be denoted by *type*(*dex*). In the second step, we describe the application of a distribution type, t, to an environment that specifies a (data) array, A, and a processor section, R; this yields a distribution of A with respect to R. The distribution type of a distribution δ will be denoted by *type*(δ).

Evaluation of Simple Distribution Expressions

- Consider a *distribution function reference dfr:*

  $$f[(a_1, \ldots, a_k)]$$

  where f is the name of a distribution function, and $a_1, \ldots, a_k$ $(k \geq 0)$ the list of its explicit actual arguments (see Sections 3.4.3 and 5.1).

  The **distribution type** associated with *dfr* is given as

  $$type(dfr) := f[(a'_1, \ldots, a'_k)]$$

  where

  - If a_i is scalar, then a'_i is the value of a_i.
  - If a_i is an array section $A(ss_1, \ldots, ss_n)$, then $a'_i = A(ss'_1, \ldots, ss'_n)$, where each $ss'_j, 1 \leq j \leq n$, is determined from ss_j by replacing each expression occurring in ss_j by its value.
  - Otherwise, $a'_i = a_i$ (i.e., the name of a_i).

Now assume that $t := f[(a'_1, \ldots, a'_k)]$ has been determined, and we want to apply t to the given environment. Then, if the ranks of A and R are n and m, respectively, f must be an n-m distribution function, with A and R as its implicit arguments. After transferring the arguments, the function is executed according to the rules specified in Section 5.1. The application must yield a distribution δ for A with respect to R.

- A *distribution extraction* $= B$ is evaluated by replacing it with $t := type(\delta^B)$. The resulting type, t, can then be applied to the given environment.
- A *distribution type name* is evaluated by replacing it with the associated distribution type, t. t can then be applied to the given environment.

Evaluation of Composite Distribution Expressions

A **composite distribution expression**, *dex*, has the form:

$$dex_1, \ldots, dex_n$$

where each dex_i is either a *simple distribution expression* yielding a 1-1 distribution type, or the elision symbol, expressing the hiding of a dimension. Assume that h is the number of elision symbols in *dex*, and $n' := n - h$. The evaluation of *dex* leaves all elision symbols unchanged, and replaces each dex_i by the associated distribution type. This yields an n'-n' distribution type, which can be applied in the context of any data array of rank n, and any processor section of rank m, where $m \geq n'$, to obtain a distribution as defined in Def.12.

3.5 Alignment Specifications

3.5.1 Syntax

1. alignment-specification → **ALIGN** aspec
2. aspec → alignment-expression | functional-alignment
3. alignment-expression → target-array-identification "(" bound-variable-list ")" **WITH** source-array-reference
4. target-array-identification → *array_name* | "$"
5. bound-variable → *variable_name* | ":"
6. source-array-reference → data-reference
7. functional-alignment → "(" alignment-function-reference ")" **WITH** source-array-section
8. alignment-function-reference → *function_reference*

3.5.2 Introduction

Vienna Fortran provides features for the construction of alignment functions (see Section 2.4). Given an alignment function α from a target array to a source array, and a distribution for the source array, then a distribution for the target array can be determined as specified in Def.11.

An **alignment specification** has the form

ALIGN *aspec*

where *aspec* is either an **alignment expression** or a **functional alignment**. Alignment expressions provide a simple syntax for specifying a class of standard alignment functions. Functional alignment, in contrast, allows references to arbitrary **alignment functions** (see Section 5.2).

3.5.3 Alignment Expressions

An *alignment-expression* is used in a context specifying one or more target arrays. We consider one such array, A, and assume that its rank is n. An alignment expression has the form:

$$ta\ (x_1, \ldots, x_n)\ \textbf{WITH}\ sar$$

where:

- ta is the **target array identification**. ta is either the name of the target array (if there is only one in the environment) or the symbol "$" (representing all arrays in the environment).
- Each $x_i, 1 \leq i \leq n$, is either a bound variable whose scope is limited by the alignment expression, or a colon. If a variable is specified, then it ranges over the i-th dimension of the index domain associated with A: it is implicitly declared of type integer, and its value range is $\mathbf{I}^A_i$. Any program variable with the same name is hidden until the end of the alignment specification. If a colon is specified, we implicitly replace it by a newly generated variable, which does not occur elsewhere in the program. For this variable, the same rules as those outlined above hold.
- sar represents the source array reference. It has the form

  $$B(ss_1, \ldots, ss_m)$$

  where B is an array name, designating an array of rank m, and the $ss_j, 1 \leq j \leq m$ are section subscripts. Each section subscript ss_j must satisfy exactly one of the three conditions specified below:

  - ss_j is a subscript triplet. In this case, none of the bound variables may occur in ss_j.
  - ss_j is a subscript that contains none of the bound variables. Then it may be an arbitrary integer expression.
  - ss_j is a subscript that contains one of the bound variables, say x_i. Then, no other section subscript $ss_{j'}, j' \neq j$, may contain x_i. Besides x_i, ss_j may contain only integer constants (including symbolic constants) and the operators "+", "-", and "*", which may be used to form expressions of type integer which are linear in x_i. Furthermore, the intrinsic functions *MAX, MIN, MOD, LBOUND, UBOUND*, and *SIZE* (see Appendix B) can be applied.

We are now in a position to specify the alignment function, α, determined by the alignment expression: Select an arbitrary tuple $\mathbf{k} = (k_1, \ldots, k_n) \in \mathbf{I}^A$, replace each bound variable x_i occurring in $B(ss_1, \ldots, ss_m)$ by k_i, and evaluate the resulting array reference, which may designate either an array element, or an array section. Let $\mathbf{I}^B(\mathbf{k})$ denote the set of indices of B associated with the evaluated reference. Then:

$$\alpha(\mathbf{k}) := \mathbf{I}^B(\mathbf{k})$$

Now, given a distribution for B, a distribution for A can be immediately constructed (Def.11).

Example 12 Alignment expressions

Consider the following set of annotated declarations:

```
REAL C(10,10,10) DIST ( BLOCK, CYCLIC, :)  TO R2
REAL D(1000)     DIST ( BLOCK)  TO R1

REAL C1(10,10,10)  ALIGN C1(I1,I2,I3)  WITH C(I1,I2,I3)
REAL C2(10,10,10)  ALIGN C2(I,J,K)     WITH C(J,I,K)
REAL C3(10,10,10)  ALIGN C3(:,J,:)     WITH C(1:5,MIN(2*J+1,10),1)
REAL C4(10,10,10)  ALIGN C4(I1,I2,I3)  WITH C(:,MAX(I2-3,1),I3)
REAL C5(5,5,5)     ALIGN C5(:,:,:)     WITH C(2::2,6,:)
REAL C6(10,10,10)  ALIGN C6(:,L,:)     WITH D(50*L+100)
REAL C7(10,10,10), C8(10,10,10) ALIGN (:,:,:) WITH D
```

Assume for this example that $\mathrm{NP1} = 2$*, i.e., the processor arrays are declared as* $R1(1:4)$ *and* $R2(1:2, 1:2)$*. Then the element-based distributions of arrays* C *and* D *are as follows:*

$\delta^C(i,j,k) = \{R2(1,1)\}$ *for* $1 \le i \le 5, j = 1,3,5,7,9$*, and all* $k, 1 \le k \le 10$
$\delta^C(i,j,k) = \{R2(2,1)\}$ *for* $6 \le i \le 10, j = 1,3,5,7,9$*, and all* $k, 1 \le k \le 10$
$\delta^C(i,j,k) = \{R2(1,2)\}$ *for* $1 \le i \le 5, j = 2,4,6,8,10$*, and all* $k, 1 \le k \le 10$
$\delta^C(i,j,k) = \{R2(2,2)\}$ *for* $6 \le i \le 10, j = 2,4,6,8,10$*, and all* $k, 1 \le k \le 10$

$\delta^D(i) = \{R1(1)\}$ *for* $1 \le i \le 250$
$\delta^D(i) = \{R1(2)\}$ *for* $251 \le i \le 500$
$\delta^D(i) = \{R1(3)\}$ *for* $501 \le i \le 750$
$\delta^D(i) = \{R1(4)\}$ *for* $751 \le i \le 1000$

The alignment expression for $C1$ *specifies the target array identification as* $C1$ *and the bound variable list* $(I1, I2, I3)$*, where* $I1$ *ranges over the first,* $I2$ *over the second, and* $I3$ *over the third dimension of* $C1$*. The range of all three variables is the set of integer numbers in the interval* $[1:10]$*. The source array reference specifies for each* $(I1, I2, I3)$ *the identical triplet* $(I1, I2, I3)$*. Thus, the associated alignment function is identity, and array* $C1$ *has the same distribution as* C*.*

The alignment expression for $C2$ *transposes the first and second dimensions, i.e., the resulting alignment function maps each index triplet* (I, J, K) *in* $\mathbf{I}^{C2}$ *to the index triplet* (J, I, K) *in* $\mathbf{I}^C$*. The distribution of* $C2$ *may be obtained from that of* C *by exchanging* i *and* j*.*

The alignment expression for $C3$ *replicates the first dimension, and collapses the third dimension, while mapping* J *to* $MIN(2 * J + 1, 10)$ *in the second dimension. Thus, the alignment function specified is*

$\alpha(i,j,k) := \{(i',j',k') \mid 1 \le i' \le 5, j' = MIN(2 * j + 1, 10)$*, and* $k' = 1\}$

From this we obtain the distribution of $C3$ *as follows:*

$\delta^{C3}(i,j,k) = \{R2(1,1)\}$ *for* $1 \le i \le 10, 1 \le j \le 4, 1 \le k \le 10$
$\delta^{C3}(i,j,k) = \{R2(1,2)\}$ *for* $1 \le i \le 10, 5 \le j \le 10, 1 \le k \le 10$

Similar considerations lead to the distributions as given below:

$\delta^{C4}(i,j,k) = \{R2(1,1), R2(2,1)\}$ *for* $1 \le i \le 10, j = 1,2,3,4,6,8,10,\ 1 \le k \le 10$
$\delta^{C4}(i,j,k) = \{R2(1,2), R2(2,2)\}$ *for* $1 \le i \le 10, j = 5,7,9$*, and all* $k, 1 \le k \le 10$

$\delta^{C5}(i,j,k) = \{R2(2,1), R2(2,2)\}$ *for* $1 \le i,j,k \le 5$

$\delta^{C6}(i,j,k) = \{R1(1)\}$ *for* $1 \le i,k \le 10, 1 \le j \le 3$
$\delta^{C6}(i,j,k) = \{R1(2)\}$ *for* $1 \le i,k \le 10, 4 \le j \le 8$
$\delta^{C6}(i,j,k) = \{R1(3)\}$ *for* $1 \le i,k \le 10, 9 \le j \le 10$
$\delta^{C7}(i,j,k) = \delta^{C8}(i,j,k) = R1$ *for all* $1 \le i,j,k \le 10$
□

3.5.4 Functional Alignment

Functional alignment allows references to arbitrary **alignment functions** (see Section 5.2). Alignment functions can be classified in a similar way as distribution functions: they are called $n - m$ functions, if they specify a mapping from an n-dimensional target array to an m-dimensional source array. Alignment

functions may contain arguments, but are not allowed to have side effects; they are referenced in the same way as FORTRAN 77 functions, except that an empty argument list need not be enclosed in parentheses. Functional alignment is specified in the form

ALIGN (*afr*) **WITH** *sas*

where *sas* is an array section representing the source array, B, of the alignment, and *afr* is a reference to an alignment function. Assume that the context in which the functional alignment occurs determines a target array, A, of rank n, and that B has rank m. Assume that the *alignment function reference, afr,* has the form

$$f[(a_1, \ldots, a_k)]$$

where f is the name of an alignment function, and $a_1, \ldots, a_k$ $(k \geq 0)$ the list of its explicit actual arguments (see Section 5.2).

Then, f must be an n-m alignment function, with A and B as its implicit arguments. After transferring the arguments, the function is executed according to the rules specified in Section 5.2. This application must yield an alignment α for A with respect to B. Then, given that B is associated with a well-defined distribution, δ^B, a distribution, δ^A is determined for A from δ^B and α, according to the construction of Def.11.

If the distribution of an array A has been defined by functional alignment in the above way, then it is associated with a sequence of arrays $\mathcal{B}= B_0, B_1, \ldots, B_r$, where

- The distribution of B_0, the **root array**, is defined only by distribution function references,
- $B_r = A$, and
- for all $j, 1 < j \leq r$, B_j is derived from B_{j-1} by functional alignment.

The distribution of B_0, together with a description of the sequence of alignment function references and array sections associated with $\mathcal{B}$, yields the **type** of δ^A. In this sequence, alignment function references are processed in the same way as distribution function references when evaluating their type (see Section 3.4.7), and array sections are characterized by the array names and the values associated with their subscripts and subscript triplets.

It can be easily seen that all alignment expressions can be eliminated, if proper alignment functions are generated. Thus, the problem of determining the type of a distribution defined via an alignment expression can be deferred to the case discussed above.

3.6 Static Array Annotations

3.6.1 Syntax

1. static-array-annotation → [distribution-specification | alignment-specification]
2. distribution-specification → **DIST** dspec
3. dspec → "("distribution-expression")" [**TO** processor-reference] | **TO** processor-reference

3.6.2 Semantics

A *static array annotation*, when appended to a declaration of one or more arrays, characterizes these arrays as statically distributed, and specifies an associated distribution. This association is valid in the program unit in which the declaration occurs, throughout all allocation instances of the arrays. Static array annotations may be appended to declarations of FORTRAN 77 arrays as well as to declarations of allocatable arrays.

All expressions occurring in components of static array annotations are evaluated at the time the declaration is evaluated. A subsequent modification of any variable occurring in the annotation has no effect on the distribution determined for the annotation.

Consider the declaration of an array A of rank n. A static array annotation, appended to such a declaration, can have one of three forms:

1. It is empty. This case is equivalent to the annotation

   DIST $(:,\ldots,:)$ **TO** R

   where the specified distribution expression contains exactly n elision symbols, and R is assumed to be the primary array.

   This denotes total replication, as explained in Section 3.4.6.

2. The annotation is a *distribution-specification*. Then we have to distinguish two cases:

   - **DIST** (*dex*) [**TO** *pref*]
     Here, *dex* is a *distribution-expression* and *pref* an optional processor reference. The meaning of this construct for the case in which *pref* is specified as a processor section has been explained in Section 3.4. If *pref* is a *processor-element-name*, $R(j_1,\ldots,j_m)$, then the rank of *dex* must be 0. In this case, *pref* is re-interpreted as the m-dimensional processor section $R(j_1:j_1,\ldots,j_m:j_m)$. If *pref* is not explicitly given, the primary processor array is substituted by default. Note that this substitution results in an error, if the condition for the dimension of the processor section specified in Def.12 is not met.
   - **DIST TO** *pref*
     This is equivalent to

     DIST $(:,\ldots,:)$ **TO** *pref*

     where the distribution expression contains exactly n elision symbols (n is the rank of A).

3. The annotation is an *alignment-specification*. The meaning of this construct has been explained in Section 3.5.

Example 13 Defaults in static array annotations

Consider the following declarations:

```
REAL E1(N)       DIST ( BLOCK)
REAL E2(N)       DIST (:)
REAL E3(N,M)     DIST ( BLOCK,CYCLIC(K))
REAL E4(N,N)     DIST ( BLOCK,:)
REAL E5(N,N)     DIST (:,:)
REAL E6(N,N)     DIST (:,:) TO $P( $NP)
REAL E7(N,N)     DIST TO R2
REAL E8(N,N)     DIST TO $P($NP)

REAL E9(N,M,L)   DIST ( BLOCK,BLOCK,BLOCK)
REAL E0(N,N)     DIST ( BLOCK,:) TO $P( $NP)
```

In the first five of these declarations, the missing processor reference is substituted by the primary array R2, which is two-dimensional. Note that this results in replication across at least one dimension of R2, if the rank of the distribution expression is less than 2.

The annotation associated with E6 specifies that all elements are to be mapped to the one processor $P($NP), while each element of E7 is owned by each processor. E8 is distributed in exactly the same way as E6.

The following are illegal:

REAL E9(N,M,L) **DIST** (*BLOCK,BLOCK,BLOCK*)
REAL E0(N,N) **DIST** (*BLOCK,:*) **TO** $P(*$NP*)

A substitution of the primary array R2 in the annotation for E9 results in an illegal annotation; and the distribution specified for E0 has rank 1 and must not be mapped to a single processor element. □

The distribution of **scalar objects** may be given by annotations that (explicitly or implicitly) specify distribution expressions of rank 0.

3.7 Dynamically Distributed Arrays

3.7.1 Syntax

1. dynamic-array-annotation → **DYNAMIC** (primary-array-annotation | secondary-array-annotation)
2. primary-array-annotation → [“,” distribution-range] [“,” initial-distribution]
3. distribution-range → **RANGE** “(”dspec-list“)”
4. initial-distribution → distribution-specification | alignment-specification
5. secondary-array-annotation → “,” **CONNECT** connection
6. connection → distribution-extraction | aspec
7. distribute-statement → **DISTRIBUTE** distribution-group
8. distribution-group → *array_name*–list “::” [(dspec | alignment-specification)] [notransfer-attribute]
9. notransfer-attribute → **NOTRANSFER** [“(”*array_name*–list“)”]

3.7.2 Dynamic Array Annotations

A *dynamic-array-annotation* characterizes the arrays in the associated declaration as dynamically distributed. Distributions for such arrays are determined at run-time, and may change during the execution of the program unit in which the declaration occurs.

We define an equivalence relation **connect** in the set of dynamically distributed arrays that satisfies the following conditions:

1. Each equivalence class consists of one distinguished member, the **primary array**, B, of the class, and 0 or more **secondary arrays**. We denote the class associated with primary array B by $\mathcal{C}(B)$.
2. The distribution of each secondary array $A \in \mathcal{C}(B)$, if any, is defined in the declaration of A by referring to B in a *secondary-array-annotation*, which specifies a *connection* by distribution extraction or alignment.
3. Distribute statements are explicitly applied to primary arrays only; their effect is to redistribute all arrays in the associated equivalence class so that the *connection* is maintained.

4. The distributions of arrays in different equivalence classes are independent of each other.

We first discuss annotations specifying primary arrays. They are of the following form:

REAL $ad_1, ad_2, \ldots, ad_r$ **DYNAMIC** [,*distribution-range*] [,*initial-distribution*]

where the $ad_i, 1 \leq i \leq r$ specify array identifiers B_i and their index domains. All B_i are declared to be **primary dynamic** arrays. If any of the two optional attributes occurs, the rank of all B_i must be the same. The meaning of the attributes is as follows:

1. **Distribution Range**
   A *distribution-range* specifies the set of all distribution types (or a superset thereof) which can be associated with the arrays B_i during the execution of the procedure in which the declaration occurs. Furthermore, the associated *processor references* may be optionally indicated [5]. The *distribution range* is specified by the keyword **RANGE**, followed by a parenthesized list of *dspec*s (see Section 3.6). The "*" can be used as a "don't care" symbol in any place where either a distribution expression or an argument for a distribution function may occur, with the meaning that it allows any legal substitution.

   Distribute statements applied to the B_i must respect the restrictions imposed by this attribute.

   If no distribution range is specified, then there is no restriction on the distributions that can be associated with a primary array.

2. **Initial Distribution**
   An *initial-distribution* of a primary array is given by a *distribution-specification* or an *alignment-specification*. It is evaluated and associated with the array each time an allocation instance is initiated.

   Note that a primary array for which an initial distribution has not been specified cannot be legally accessed before it has been explicitly associated with a distribution by the execution of either a distribute statement or a procedure call.

We now proceed to the specification of secondary arrays. Let

REAL $ad_1, ad_2, \ldots, ad_s$ **DYNAMIC, CONNECT** *connection*

denote the annotated declaration of secondary arrays A_j, $1 \leq j \leq s$, where *connection* can be any of the following:

- a distribution extraction: $= B[.dim]$
- an alignment expression: ... **WITH** $B(\ldots)$
- a functional alignment: (afr) **WITH** $B(\ldots)$

In all three cases, all secondary arrays A_j are **connected** to an array B, which must be a primary array. B is called the **source array** of the connection. As a result of this declaration, the A_j are entered into the equivalence class $\mathcal{C}(B)$. Each time a distribute statement is applied to B, the distribution of all A_j is newly defined such that δ^{A_j} and δ^B remain in the relationship established by the connection.

The same primary array may be the source array for more than one secondary array annotation.

[5] If the *processor references* are not given, any processor sets may be associated with the array; that is, there is no default rule substituting the primary processor array in this case.

Example 14 Dynamic array annotations

```
REAL B1(M)  DYNAMIC
REAL B2(N) DYNAMIC, DIST( BLOCK)
REAL B3(N,N), B4(N,N)  DYNAMIC,
&                RANGE (( BLOCK,BLOCK), (CYCLIC, CYCLIC(*)),(*,CYCLIC)),
&                DIST( BLOCK,CYCLIC) TO R2(::2,::4)
REAL A1(N,N), A2(N,N) DYNAMIC, CONNECT (=B4)
```

All arrays introduced in the above declaration are dynamically distributed. $B1$ is a primary array with unspecified distribution range and no initial distribution. $B2$ is a primary array with an unspecified distribution range and the initial distribution ($BLOCK$). $B3$ and $B4$ are primary arrays for which a distribution range as well as an initial distribution are specified.

$A1$ and $A2$ are declared as secondary arrays, which are connected to $B4$ via distribution extraction. As a consequence, $\mathcal{C}(B4) \supseteq \{B4, A1, A2\}$; the connection specifies that the distribution type of $A1$ and $A2$ will be always the same as that of $B4$. □

3.7.3 Distribute Statements

Consider a *distribute-statement* of the form

DISTRIBUTE B:: *da* [*notransfer-attribute*]

where the B is an array name associated with a primary array, and *da* is either empty, or a *dspec* (Section 3.6), or an *alignment-specification* (Section 3.5). The allocation status of B must be **allocated**.

da is evaluated at the time the distribute statement is executed; its meaning is the same as specified in Section 3.6. Note that components of *da* such as section-subscripts in a processor reference or arguments for distribution or alignment functions may depend on variables whose actual values are used in this step. Let $\hat{\delta}_1$ denote the distribution state immediately before the execution of the distribute statement. The execution of this statement has then the following effect:

1. Evaluate *da* and apply the resulting distribution type, t, to B: this yields a distribution, δ^B, for array B.
2. For each array $A \in \mathcal{C}(B) - \{B\}$, determine a distribution from t, $\mathbf{I}^A$, and the *connection* between A and B, as established in the associated secondary array annotation. Let δ^A denote the resulting distribution.
3. We define a set $NOTRANSFER$ as follows:
   (a) If a *notransfer-attribute* is not specified, then $NOTRANSFER := \phi$
   (b) If a *notransfer-attribute* is specified in the form **NOTRANSFER**, then $NOTRANSFER := \mathcal{C}(B)$.
   (c) If a *notransfer-attribute* is specified in the form

   NOTRANSFER$(C_1, \ldots, C_m)$

   where $m \geq 1$, then all C_j, $1 \leq j \leq m$ must be elements of $\mathcal{C}(B)$, and we define $NOTRANSFER := \{C_1, \ldots, C_m)\}$.
4. A new distribution state, $\hat{\delta}_2$, is derived from $\hat{\delta}_1$ by associating B with δ^B, and all $A \in \mathcal{C}(B) - \{B\}$ with δ^A, as computed above. For all arrays in the set $NOTRANSFER$, only the access function is changed and the elements of the arrays are not physically moved.

A distribute statement with more than one primary array in its *distribution-group*:

$$\textbf{DISTRIBUTE } B_1, \ldots, B_r :: da \ [\ \textbf{NOTRANSFER}\,[(ntl)]]$$

is equivalent to the sequence

$$\begin{array}{c}\textbf{DISTRIBUTE } B_1 :: da \ [\ \textbf{NOTRANSFER}\,[(ntl_1)]] \\ \ldots \\ \textbf{DISTRIBUTE } B_r :: da \ [\ \textbf{NOTRANSFER}\,[(ntl_r)]]\end{array}$$

where the ntl_i are sublists of ntl, specifying exactly those elements in ntl that belong to $\mathcal{C}(B_i)$.

Example 15 *We refer to the declarations in the previous example. It is assumed that the statements below are executed unconditionally in the order of their appearance in the text.*

DISTRIBUTE B1 :: (*BLOCK*)
...
K= *expr*
DISTRIBUTE B1,B2 :: (*CYCLIC*(K))
...
DISTRIBUTE B3 :: (*BLOCK,CYCLIC*)
DISTRIBUTE B4 :: (=B1, *CYCLIC*(3))
...

In the first statement, the array B1 is distributed by (BLOCK).
In the second statement, B1 and B2 (both of which are currently distributed by (BLOCK)) are redistributed as (CYCLIC(K′)), where K′ denotes the value assigned to the variable K in the assignment K = expr.
The third statement redistributes B3 as (BLOCK,CYCLIC); in the next statement, B4 and the associated secondary arrays A1 and A2 are distributed as (CYCLIC(K′),CYCLIC(3)).

Example 16 Indirect distributions

INTEGER C(M) **DIST** (*BLOCK*)
REAL A(M) **DYNAMIC**
...
DISTRIBUTE A :: *INDIRECT*(C)

This example illustrates an application of the INDIRECT intrinsic distribution function (see Section 3.4.3). The mapping array C is a statically distributed array, and distributed by (BLOCK). At the time the distribute statement is executed, all elements $C(\mathbf{i}), \mathbf{i} \in \mathbf{I}^A$ *must be defined. Their values determine the processor indices to which the corresponding elements of A are to be mapped.*□

3.8 Control Constructs

3.8.1 Syntax

1. control-construct → dcase-construct | if-construct
2. dcase-construct → select-dcase-statement condition-action-pair... end-select-statement
3. select-dcase-statement → **SELECT DCASE** "(" *array_name*–list ")"
4. condition-action-pair → **CASE** condition action
5. condition → query-list | **DEFAULT**

6. query → [name-tag] (dspec | "*")
7. name-tag → *array_name* ":"
8. action → [*executable_statement*]...
9. end-select-statement → **END SELECT**
10. if-construct → *logical_if_statement* | *block_if_statement* | *else_if_statement*

3.8.2 Introduction

Consider a reference, *aref*, to an array A. The distribution type associated with A at the time that reference is encountered during program execution may or may not be derivable from an analysis of the program text. For example, if A is associated with a declaration annotation that specifies a distribution function reference whose arguments are all known, then the distribution type of *aref* is known as well. In contrast, if A is a dynamically distributed array, or a dummy argument with an inherited distribution (see Section 3.10) then it may not be possible to determine the distribution type of *aref* from the context.

The *control-construct*s have been included in the language to alleviate the problems arising from the second case: first, they allow the user to formulate an algorithm, depending on the actual distribution type of one or more arrays; secondly, they provide the compiler with information about the distribution of arrays. They include the *dcase-construct*, which is modeled after the Fortran 90 CASE construct, and the *if-construct*, which is based on a generalized form of *logical_expression*s, and the related Fortran if statements.

3.8.3 The DCASE Construct

The dcase-construct has the form

SELECT DCASE $(A_1, \ldots, A_r)$ $cap_1, \ldots, cap_m$ **END SELECT**

where

- $r \geq 1$ and all A_i, $1 \leq i \leq r$, are array names. The A_i are called **selectors**. At the time of execution of the dcase construct, each selector must be allocated and associated with a well-defined distribution.
- $m \geq 1$ and each cap_j, $1 \leq j \leq m$, is a *condition-action-pair*, where the *condition* is either a *query-list* or the keyword **DEFAULT**, and the *action* is a *block*. A *block* is a sequence of *executable_statement*s, including the statements of the language extension, except for the distribute statement. None of the statements in a block may be the target of a branch from outside of that block. It is permissible to branch to an *end-select-statement* only from within the dcase construct.

The dcase construct selects for execution at most one of its constituent blocks. It is evaluated as follows:

1. The distribution of each selector, and its type, are determined.
2. Let $(c_1, a_1), (c_2, a_2), \ldots$ denote the sequence of condition action pairs in the dcase construct. Then $c_1, c_2, \ldots$ are sequentially evaluated until either a j, $1 \leq j \leq m$ is reached such that c_j **matches**, or no match occurs.

   If c_j matches, then the associated action a_j is executed. This completes the execution of the dcase construct. If no match occurs, the execution of the dcase is completed without executing an action.

A condition c_j **matches** iff one of the following constraints is satisfied:

- c_j is the keyword **DEFAULT**

- c_j is a list of queries, each of which matches. Each query tests the distribution of one selector array.

  Query lists may be either **positional** or **name-tagged**.

  In a positional query list, the queries are associated with the selectors $A_1, A_2, \ldots$ in this order. In this case, none of the queries can have a *name-tag* attached to it.

  In a name-tagged query list, the selector associated with each query is explicitly specified by a *name-tag*. The order in which the queries occur in such a list is semantically irrelevant.

  A query list need not contain a query for every selector specified in the *select-dcase-statement*. In such a case, an implicit "*" is inserted for every selector which is not represented.

  Now consider a single query.

  The query specifies a *modified distribution expression*, *dex*, and an optional *processor-reference*, *pref*. If *dex* and *pref* are both specified, then the query matches iff both *dex* and *pref* match. If *pref* is not specified, then the query matches iff *dex* matches[6].

  A *processor reference pref* **matches** iff it specifies the set of processors, in the same order, to which the selector array has been distributed.

  The *modified distribution expression* can take any of the following forms:

  - the symbol "*"
  - a distribution expression without distribution extractions
  - a modified distribution expression, without distribution extractions, which may contain an asterisk, "*", as a dimensional expression or in an argument position of a reference to a distribution function.

  Assume that the query is associated with a selector of distribution type t.

  1. If *dex* = "*", then *dex* matches.
  2. If *dex* is a *distribution-expression*, and has distribution type t', then *dex* matches iff $t = t'$.
  3. If *dex* does not satisfy the two conditions given above, then it matches iff $t = t$", where t" is the result of substituting for each "*" occurring in *dex* the corresponding component of t. This substitution must be possible and well-defined.

Example 17 The dcase construct

```
REAL B1(M)  DYNAMIC
REAL B2(N) DYNAMIC, DIST( BLOCK)
REAL B3(N,N), DYNAMIC,
&             RANGE (( BLOCK,BLOCK), (CYCLIC, CYCLIC(*)),(*,CYCLIC)),
&             DIST( BLOCK,CYCLIC)

    ...

DISTRIBUTE B1 ...
DISTRIBUTE B2 ...
    ...
DISTRIBUTE B3 ...
```

[6] Note that there is no default rule here – in contrast to *distribution-specifications*– that substitutes the primary processor array, if *pref* is not given.

...

SELECT DCASE (B2)
  CASE (*BLOCK*)
    a_1
  CASE (*CYCLIC*)
    a_2
  CASE (*CYCLIC(*)*)
    a_3
  CASE *
    a_4
END SELECT

In this dcase construct, we use one selector, the array B2. Depending on the actual distribution associated with B2 at the time the dcase construct is executed, exactly one of the four statement blocks a_1,...,a_4 will be executed: Let t denote the distribution type associated with δ^B. Then, if $t = (BLOCK)$, the first query list matches, and a_1 is executed.

If $t = CYCLIC$ (note that this is equivalent to $CYCLIC(1)$), then the first query list fails, but the second matches, and a_2 is executed.

If $t = CYCLIC(L)$, where $L > 1$, then the first two query lists fail, and the third matches. As a consequence, a_3 will be executed.

Finally, if t is of any other distribution type, then the first three query lists fail, and the fourth matches. In this case, a_4 is selected for execution.

The slightly more complicated example below uses a selector set with the three members $B1, B2$, and $B3$. Assume that DF is a distribution function with two integer arguments, and that t_i is the distribution type associated with B_i:

SELECT DCASE (B1,B2,B3)
  CASE (*BLOCK),(BLOCK),(CYCLIC(2),CYCLIC)*)
    a_1
  CASE B1: (*DF(L1,L2)*), B3:(*BLOCK,**))
    a_2
  CASE (*DF(*,*)),(DF(*,*)*)
    a_3
  CASE B3:(*BLOCK,CYCLIC*)
    a_4
  CASE DEFAULT
    a_5
END SELECT

The first query list is positional; it matches if $t_1 = t_2 = (BLOCK)$, and $t_3 = (CYCLIC(2), CYCLIC)$.
The second list is name-tagged; it matches if $t_1 = (DF(L1, L2))$, $t_3 = (BLOCK, t')$, where t' is arbitrary (within the constraints set by the range attribute for this dimension), and t_2 is any distribution type.
The third query list matches if $t_1 = (DF(a_1, a_2))$ and $t_2 = (DF(a_3, a_4))$, where the a_i are arbitrary. t_3 may be any distribution type.
The fourth query list matches if $t_3 = (BLOCK, CYCLIC)$. t_1, t_2 are irrelevant in this case.
Finally, the fifth query list is always matched. Thus, if none of the first four query lists match, then a_5 will be executed.□

3.8.4 The IF Construct

The *if-construct* of Vienna Fortran includes the Fortran statements

- *logical_if_statement*
- *block_if_statement*
- *else_if_statement*

all of which are based upon a **generalized logical expression**. A generalized logical expression is a Fortran *logical_expression*, which, in addition may contain references to the intrinsic functions IDT and $IDTA$. These functions perform a test of the distribution types associated with their arguments and, optionally, of the processor sections to which the arguments are distributed; they yield a logical value. We now describe their meaning.

IDT is an acronym for "**I**dentical **D**istribution **T**ypes". A reference to IDT has the form

$$IDT(A[,dim],q)$$

where A denotes an array, the optional argument dim a dimension of A, and q is a *query* without a *name tag*. (see Section 3.8.3).

The reference is evaluated as follows: First, t is determined as the type associated with the *distribution extraction* $= A.dim$ or $= A$, depending on whether or not dim is specified (see Section 3.4.4). The reference to IDT yields **true** iff t and q match according to the rules specified in Section 3.8.3.

Example 18 IF Construct

```
REAL B1(M),B2(N),B3(N,N) DYNAMIC

  ...

DISTRIBUTE B1 ...
DISTRIBUTE B2 ...
DISTRIBUTE  B3 ...

  ...

IF (IDT(B2,(BLOCK))) THEN
  a1
ELSE IF (IDT(B2,(CYCLIC))) THEN
  a2
ELSE IF (IDT(B2, CYCLIC(*))) THEN
  a3
ELSE
  a4
END IF

  ...

IF (IDT(B3,(BLOCK, CYCLIC(*))) .AND. (X .LT. Y)
&                              .AND. .NOT. (IDT(B2, INDIRECT(*))))
  a3
END IF
```

The first IF statement has the same effect as the first dcase construct used in Example 17.□

IDTA is an acronym for "**I**dentical **D**istribution **T**ypes of **A**rrays" and provides a special syntax for the case where the distribution types of two arrays are compared. A reference to *IDTA* has the form

$$IDTA(A_1[,dim_1],A_2[,dim_2])$$

where A_1 and A_2 denote arrays, and dim_1 and dim_2 are optional arguments denoting a dimension of the associated array. The effect of a reference $IDTA(A_1[,dim_1],A_2[,dim_2])$ is equivalent to the effect of $IDT(A_1[,dim_1],(=A_2[.dim_2]))$.

3.9 Allocatable Arrays

3.9.1 Syntax

1. allocatable-array-declarator → *array_name* "(" deferred-shape-spec-list ")" allocatable-attribute
2. deferred-shape-spec → ":"
3. allocatable-attribute → **ALLOCATABLE**
4. allocatable-array-annotation → actual-array-annotation
5. allocate-statement → **ALLOCATE** "("allocation-list")"
6. allocation → *array_declarator*
7. deallocate-statement → **DEALLOCATE** "("*array_name*–list")"

3.9.2 Semantics

An **allocatable array** is an array declared with the *allocatable-attribute*, which is written as **ALLOCATABLE** . The declaration of an allocatable array defines the rank of the array, but not its index domain, by a *deferred-shape-spec-list* which contains exactly one colon for each dimension. For example, an allocatable array A of rank 2 can be declared in the form

REAL $A(:,:)$ **ALLOCATABLE**

The declaration of an allocatable array may be associated with an *actual array annotation*.

A static array annotation introduces a statically distributed array. The annotation is evaluated at the time the declaration is evaluated; the resulting distribution type and processor reference are associated with every allocation instance of that array, and must remain invariant within an allocation instance.

A dynamic array annotation designates the allocatable array as dynamically distributed. If a range attribute or an initial distribution is specified, then the associated distribution expressions are evaluated each time an allocation instance is created for the array.

An allocation instance for an allocatable array is created by an *allocate-statement*, and terminated by a *deallocate-statement*. The allocate statement has the form:

ALLOCATE $(ad_1,\ldots,ad_r)$

where all ad_i are *array_declarator*s, excluding *assumed size array declarators*. At the time the allocate statement is executed, the values of the lower and upper bound expressions in each array declarator determine the index domain for the allocation instance of the associated array. The allocation status of each array changes from **deallocated** to **allocated**. A subsequent redefinition or undefinition of any entities in the bound expressions does not affect the index domain.

The execution of the allocate statement for an array A results in the association of A with a distribution iff either A is statically distributed, or A is dynamically distributed, with an initial distribution specified in the declaration.

The deallocate statement has the form

$$\textbf{DEALLOCATE}\,(A_1, \ldots, A_m)$$

where the A_i are array names associated with allocatable arrays. At the time of execution of the deallocate statement, all arrays A_i must have the allocation status **allocated**. The effect of the execution of the deallocate statement is the termination of the allocation instance associated with each of the A_i; their allocation status becomes **deallocated**.

3.10 Procedures

3.10.1 Syntax

1. dummy-array-annotation → actual-array-annotation [dummy-annotation-attribute]... | inherit-annotation
2. inherit-annotation → **DIST** "(" "*" ")" ["," distribution-range]
3. dummy-annotation-attribute → restore-attribute | nocopy-attribute | notransfer-attribute
4. restore-attribute → **RESTORE**
5. nocopy-attribute → **NOCOPY**

3.10.2 Semantics

The dummy array arguments in a procedure can be distributed in a manner similar to actual arrays so as to specify how the arrays will be viewed and accessed within the procedure. In addition, local arrays may either be distributed explicitly or aligned with a dummy argument. While Vienna Fortran usually accesses dummy array arguments by the standard Fortran call-by-reference paradigm, there are situations (e.g. when an array section is transferred and redistributed) in which a copy in/copy out semantics must be adopted. It is also adopted in any situation where there is not enough information to decide whether reference transmission is semantically valid. The user may override this by specifying the *nocopy attribute*, as described below.

Any activation of a procedure q implicitly transfers the whole set of processors to that procedure. If a processor declaration occurs in the specification part of q, then:

1. For any processor array R that is specified by an assumed shape array declarator of rank n, a processor array R' of rank n must be uniquely determined in the calling procedure. The shape of R is defined by the shape of R'.
2. If a processor array R is specified by an adjustable array declarator of rank n, then either a processor array, R', of rank n, which determines its shape, must be uniquely determined in the calling procedure, or R is uniquely defined as a reshape of another specified processor array R'' whose shape is known.

The rules governing the declaration and use of dummy arguments in a procedure are specified below:

1. All *actual array annotations* as described in Section 3 can be used, with the same semantics. Their scope is limited to the procedure.

2. If the dummy argument has a static distribution then this distribution is enforced upon procedure entry, i.e., the array may have to be redistributed to match the specified distribution. If the actual argument is also statically distributed, and the corresponding dummy argument has been redistributed at procedure entry, then the original distribution is restored on procedure exit. If the actual argument is dynamically distributed, then no such restoration is required. Restoration of the original distribution can always be enforced by using the *restore attribute* specified by the keyword **RESTORE**, either in the dummy argument annotation or with the actual argument at the point of the procedure call.

3. If the dummy argument is declared as dynamically distributed and no initial distribution is given, then its distribution upon procedure entry is defined by the distribution of the actual argument (which may be undefined). If an initial distribution is specified, then as in the case of statically distributed dummy arguments, a redistribution may be required to enforce this initial distribution. A dynamically distributed dummy argument may be a target of an explicit distribute statement within the procedure. The original distribution is restored if the actual argument is statically distributed or if the *restore attribute* has been specified.

4. For dummy arguments which are either statically distributed or dynamically distributed with an initial distribution, a *notransfer attribute* (using the keyword **NOTRANSFER**) can be specified with the associated dummy array annotation. In such a situation, if a redistribution is required at procedure entry, then only the access function is changed and the elements of the array are not physically moved. This attribute is useful when the values of the dummy argument are first going to be defined in the procedure before being used. It is also appropriate if a dynamically distributed array has not been distributed before the procedure call.

5. A *nocopy attribute* can be attached to the declaration of a dummy argument to suppress the generation of a procedure-local copy of the dummy array. In this case, the argument transmission is by reference, i.e., the actual argument is directly accessed within the procedure. For statically distributed dummy arguments, it is an error if the distributions of the actual and the dummy argument are not identical. For dynamically distributed dummy arguments the actual argument may not be statically distributed. Note that a *restore attribute* is not permitted with the *nocopy attribute*. The *nocopy attribute* is specified by the keyword **NOCOPY**.

6. In addition to the explicit distributions described above, a dummy argument may inherit the distribution of the corresponding actual argument. This can be specified using the annotation **DIST(*)** with the declaration. The corresponding actual argument may have different distributions on different call statements and no implicit redistribution takes place on procedure entry. The set of all possible argument distributions can be specified by an optional *distribution range attribute* as used in the dynamic array annotation. A dummy argument which inherits its distribution may *not* be a target of a distribute statement even if the corresponding actual argument is dynamically distributed.

7. If an actual argument is dynamically distributed and is a member of a connect set, the association is temporarily "disconnected" during the execution of the procedure. The connection is "restored" on exit from the routine. That is, if the actual argument was a secondary array, it may have to be redistributed to reestablish its alignment with the primary array. If the actual argument was a primary array and if it has a different distribution before and after the call, then the associated secondary arrays have to be redistributed after exit from the routine to maintain the connection.

The fundamental issue here is that the declarations within a procedure provide a local view of the arrays which is independent of the distribution of the actual argument. That is, every reference to the array locally within the procedure is interpreted according to the local declaration. Thus if a dummy array is declared with a static distribution then it cannot be a target of a distribute statement within the procedure even if the actual argument has been dynamically distributed in the calling procedure. Similarly, if a dummy

Actual Argument **Dummy Argument**	Static	Dynamic	Dynamic + *restore*	Dynamic (undistributed)
Static	1	2	1	3
Static + *restore*	1	1	1	Error
Static + *notransfer*	4	5	4	6
Static + *notransfer* + *restore*	4	4	4	Error
Static + *nocopy*	7	7	7	Error
*	8	8	8	Error

1: *Redistribute on entry if necessary to match local declaration and restore original distribution on exit.*

2: *Redistribute on entry (if necessary), no redistribution on exit.*

3: *Distribute on entry to match dummy argument but do not restore on exit.*

4: *Redistribute on entry (if necessary) without moving data values and restore distribution on exit (including the moving of data values).*

5: *Redistribute on entry (if necessary) without moving data values, no redistribution on exit.*

6: *Distribute on entry without moving data values, no redistribution on exit.*

7: *Error if distributions of dummy and actual arguments are not identical. No redistributions required.*

8: *No redistribution on entry or exit. Dummy argument cannot be a target of a* **distribute** *statement.*

Note: "Static" and "Dynamic" denote arguments which are declared with static and dynamic distributions respectively.

Table 1: Actions on procedure entry and exit: Statically distributed dummy arguments

Actual Argument **Dummy Argument**	Static	Dynamic	Dynamic + *restore*	Dynamic (undistributed)
Dynamic	1	2	1	3
Dynamic + *restore*	1	1	1	Error
Dynamic + *notransfer*	4	5	4	6
Dynamic + *notransfer* + *restore*	4	4	4	Error
Dynamic + *nocopy*	Error	2	1	3

1: *Redistribute on entry if necessary to match the initial distribution of the dummy argument and restore distribution on exit.*

2: *Redistribute on entry (if necessary), no redistribution on exit.*

3: *Distribute on entry (if necessary), no redistribution on exit. Error if dummy argument is not distributed before use.*

4: *Redistribute on entry (if needed) without moving data values and restore distribution on exit.*

5: *Redistribute on entry (if needed) without moving data values, no redistribution on exit.*

6: *Distribute on entry (if necessary) without moving data values, no redistribution on exit. Error if dummy argument is not distributed before use.*

Note: "Static" and "Dynamic" denote arguments which are declared with static and dynamic distributions respectively.

Table 2: Actions on procedure entry and exit: Dynamically distributed dummy arguments

array is declared to be dynamically distributed then it can be redistributed locally even if the actual array is statically distributed. The original distribution is restored according to the above stated rules.

Tables 1 and 2 provide a detailed picture of the actions to be taken on procedure entry and exit for various combinations of actual and dummy argument declarations. The columns specify the possible actual argument attributes while the rows detail the dummy argument attributes. Note that an actual argument which is static does not need a restore attribute since the semantics of the language require that the distribution of the array be the same before and after the procedure call. The last column denotes the situation in which the actual argument is dynamic and has not been distributed before the procedure call.

Table 1 gives the associated actions for dummy arguments with static distributions while Table 2 is for dynamic dummy arguments. Note that the *notransfer attribute* holds meaning for a dynamic distribution only when it is supplied with an initial distribution.

Example 19 Array arguments

A number of different situations may arise in conjunction with the transfer of distributed arrays to procedures. Some of these are exemplified in the following code fragment:

```
PARAMETER (N=2000)
REAL A(N) DIST (CYCLIC)
REAL B(N) DIST (CYCLIC)
REAL C(N) DYNAMIC
   ⋮
CALL SUB(N,A,B,C)
   ⋮


SUBROUTINE SUB(N,A1,B1,C1)
REAL A1(N) DIST (*)
REAL B1(N) DIST (BLOCK)
REAL C1(N) DIST (BLOCK) NOTRANSFER

REAL D1(100) DIST (=A1)
INTEGER E1(500) DIST (BLOCK)
```

Here, array A1 inherits the distribution of array A; it is not copied. Upon entry to the procedure, array B must be redistributed: since B is statically distributed, the original distribution must be restored when the procedure is exited. The dynamically distributed array C is the actual argument of the (statically distributed) dummy argument C1: if C has does not have a block distribution, it is redistributed on entry but no actual values are transferred. There is no redistribution on exit, so C will subsequently have a block distribution. If a restore attribute *had been specified in the subroutine call as follows:*

```
CALL SUB(N,A,B,C::RESTORE)
```

or in the declaration of C1 as follows:

```
REAL C1(N) DIST (BLOCK) NOTRANSFER, RESTORE
```

then the distribution of C before the call would be restored.

The distribution type of the local variable D1 is extracted from the distribution of the dummy argument A1, and will thus be CYCLIC *in this incarnation of the procedure. Local array E1, in contrast, will always have a block distribution.* □

3.11 Common Blocks

Common blocks permit different program units within a Fortran program to define and reference the same data without using arguments. They also allow the sharing of storage units across program units. The association of variables and arrays in separate instances of a common block is by storage rather than by name based on a storage sequence defined for common blocks.

In Vienna Fortran, the user may retain the FORTRAN 77 semantics for common blocks by not distributing any of the objects in any common block with that name in any program unit. In this case, the common block storage sequence and association mechanism is maintained. Further, any object in the program's data space which is not explicitly distributed may be equivalence associated with an area of such a common block according to FORTRAN 77 rules.

However, Vienna Fortran also permits explicit distribution of arrays within named common blocks with the following restrictions and modification of the common block storage concept. Note that objects in blank common are not allowed to be distributed, thus, following sequential FORTRAN 77 semantics.

Individual arrays in a common block may be distributed explicitly to processors in the same way as other arrays by declaring them with a distribution annotation. However, such arrays are allowed to be distributed statically only. Also, following Fortran 90, arrays in common blocks may not be allocatable.

Objects in a common block which are not explicitly distributed are replicated across the processors. A sequence of such entities in a common block is called a *replicated section* of the common block.

Thus, a named common block consists of a set of replicated sections which may be interspersed with distributed arrays. Common blocks with the same name must have the same sequence of replicated sections and distributed objects. This provides an association between the replicated sections and distributed objects across the different occurrences of the same common block.

If one or more arrays in a common block are explicitly distributed, then the common block storage sequence as a whole is not maintained. That is, the storage allocated on a processor for a common block may or may not be contiguous. However, the size of corresponding replicated sections in all common blocks with the same name must be the same and the usual common block storage sequence holds within it.

Similarly, corresponding distributed arrays in all the common blocks with the same name have the following restrictions. They must be of the same size and their types must require the same storage units. Such arrays must be explicitly distributed in each program unit in which the common block is specified. However, along with the usual static distribution annotations, arrays in common blocks can inherit the distribution. This can be done by specifying **DIST(*)** as the distribution in a manner similar to that used for dummy arguments (see Section 3.10). When using such an implicit distribution, at least one program unit in which the common block occurs must have an explicit static distribution specified for the distributed array. This then becomes the distribution of all the corresponding distributed arrays.

When explicit distribution annotations are specified in more than one location for corresponding distributed arrays, then the distribution specification, the rank and the shape of all the specifications must agree.

Equivalence association with other data objects is permitted only for data within a replicated section of a common block. Following FORTRAN 77, it may extend beyond that replicated section only if the replicated section occurs at the end of the common block. Data objects associated with storage space in a replicated section may not be themselves distributed.

Example 20 Common block usage

The first common block, shown below, does not contain any data explicitly distributed by the user. As a consequence, the data in common blocks with the same name may be used in the usual FORTRAN *77 manner.*

```
      PROGRAM MAIN
        ...
      COMMON /COM1/ X, Y(12), B(12,30), A, AZ, AX
C     NONE OF THESE ITEMS ARE DECLARED
```

In contrast, several objects in the following common block are explicitly distributed:

```
      PROGRAM MAIN

      REAL A(12) DIST( BLOCK)
      REAL B(4,5) DIST( CYCLIC,:)
        ...
      COMMON /COM2/ CC, DD, EE, FF, GG, HH, A, B
```

Arrays A *and* B *are distributed explicitly and thus determine the distribution of these two storage areas in the common block. The variables in the common block before them comprise a replicated section of the common block and they will be stored contiguously. In a subroutine of the same program, a common block with the same name may be declared with:*

```
      REAL S(4,3) DIST(*)
      REAL T(2,5,2) DIST(*)
        ...
C     THIS IS PERMITTED
      COMMON /COM2/ R(6), S(4,3), T(2,5,2)
```

The array R *is not declared separately in the subprogram; it will be associated with the six variables of the replicated section above. The arrays* S *and* T *are declared such that they inherit their distributions from the distributed common objects, named* A *and* B *above, respectively, with whom they are associated by storage.*

However, the following declaration of /COM2/ *in a subroutine is not permitted:*

```
      REAL E(6) DIST( BLOCK)
      REAL Z(2,5,2) DIST(:, CYCLIC,:)
        ...
C     THIS IS NOT PERMITTED
      COMMON /COM2/ E, X(8), Y(4), Z
```

Here, the replicated section of COM2 *has been associated with an explicitly distributed object. Secondly, an attempt has been made to associate both arrays* X *and* Y *with the first distributed common object. Finally, the second distributed common object of* COM2 *is redistributed by the explicit distribution of array* Z*. All three manipulations are not permitted.* □

Equivalence Association

No distributed array may be associated by equivalence with any other distributed object. Equivalence associated is permitted between replicated data only.

4 FORALL Loops

4.1 Syntax

1. forall-loop → forall-statement private-var-decls forall-block end-forall
2. forall-statement → label-forall-statement | nonlabel-forall-statement
3. label-forall-statement → [forall-construct-name ":"] **FORALL** *label* forall-control
4. nonlabel-forall-statement → [forall-construct-name ":"] **FORALL** forall-control
5. forall-control → (control-variable | "(" control-variable-list ")") [on-clause]
6. control-variable → *variable_name* "=" *integer_expr* "," *integer_expr* ["," *integer_expr*]
7. on-clause → **ON** processor-element-name
8. processor-reference → **OWNER** "(" *array_element_name* ")" | processor-element-name
9. private-var-decls → *dimension_statement* | *type_statement*
10. forall-block → allocate-statement | deallocate-statement | reduction-statement | *executable_statement*
11. end-forall → end-forall-statement | *continue_statement*
12. end-forall-statement → **END FORALL** [forall-construct-name]
13. reduction-statement → **REDUCE** "(" reduction-op "," *variable* "," *expression* ["," order] ")"
14. reduction-op → **SUM** | **MULT** | **MAX** | **MIN** | *function_name*
15. order → **LEFT** | **RIGHT** | **TREE**

4.2 Semantics

As another extension to Fortran, Vienna Fortran supports explicitly parallel loops called *forall-loop*s. 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 (read-write) inter-iteration dependencies. 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. Note that the same data item can be read by two iterations.

The form of the simple forall loop header is as follows:

FORALL [*label*] *variable_name* = *integer_expr*, *integer_expr* [, *integer_expr*]

This is the same as the FORTRAN 77 do loop header except we restrict the initial, termination and increment expressions to be of type integer. Also, as in the Fortran 90 do construct, Vienna Fortran allows a block format forall loop which uses an **END FORALL** statement instead of a label to specify the end of the loop.

There is an implied synchronization at the beginning and the end of a forall loop. That is, all processors executing the iterations of the loop synchronize before the start of the loop. Similarly, they all synchronize at the end of the loop before continuing with the execution of the statements after the loop. The compiler can optimize the code by removing redundant synchronizations as long as the semantics is maintained.

Example 21 Forall loop with indirect reference

Forall loops are particularly useful in situations where the compiler cannot detect the data independence of loop iterations. For example, in the code segment shown below an index array, X, is used to assign values to another array, A.

```
      FORALL 10 I = 1, N
            A(X(I)) = ...
               ⋮
10    CONTINUE
```

Here, if the values of the array X are unique then the execution of the iterations will not create a write-write conflict. However, since the compiler cannot check this, the user by specifying a forall loop instead of a do loop asserts that the iterations are in fact independent. Forall loops whose iterations are not independent are illegal and the results from executing such programs are undefined. Note that a data item assigned within a loop iteration can legally be accessed within the same loop iteration before and after the assignment resulting in its old and new value respectively. □

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 iteration. This allows the compiler to optimize the communication required for the loop by overlapping communication with computation and also by combining messages between two processors arising from different loop iterations.

Private Variables

Vienna Fortran allows private variables in forall loops. The declaration of the variables follows the loop header and precedes any executable statement. The scope of the private variables is the forall loop in which they are declared and semantically each iteration gets its own copy of the declared variables. Thus, a private variable is undefined at the start of each iteration and is not accessible outside the loop. If the name of a private variable is the same as a name declared outside the forall loop, the local declaration hides the outer declaration till the end of the loop.

Private variables can be scalars or arrays including allocatable arrays. However, no common blocks can be declared within a forall loop. Distribution annotations are not allowed with such declarations since the variables are private to an iteration and hence will be allocated on a single processor.

Example 22 Private Variables in a Forall loop

```
      FORALL I = 1, N
        INTEGER K
        REAL X(100)
          DO 10 K = 1, 100
            X(K) = ...
10      CONTINUE
            ⋮
      END FORALL
```

In the above code segment, K and X are declared private to the forall loop. Logically there are N copies of the integer K and the array X, one for each iteration of the loop. Thus, assignments to such variables does not cause inter-iteration dependences as would be the case if, for example in the above code, K was declared outside the loop. Since the local variables are defined within each iteration, the compiler can optimize storage by reusing space for local variables where possible. For example, if the above forall loop is to be executed on \$NP *processors, then instead of N locations, the compiler can allocate only* \$NP *storage locations (one per processor) for the integer K.*□

Loop Body

The loop body of a forall loop can consist of any legal FORTRAN 77 executable statement as long as the restriction on no read-write inter-iteration dependences is observed. This includes the allocate and deallocate statements discussed in Section 3.9 and the reduction statement discussed in the next subsection 4.4 below.

As in the case of FORTRAN 77 do loops, none of the statements in the loop body of a forall loop may be the target of a branch from outside of that body. However, unlike FORTRAN 77, control cannot be transferred from within the loop to outside the loop. Also, procedures called from within a forall loop body are restricted in the following way. FORTRAN 77 allows sequence association in that an array dummy argument can be associated with an actual argument which is a single array element. Vienna Fortran does not allow this, thus forcing array dummy arguments to be associated with actual array arguments only. Note that this restriction is placed only on procedures called from forall loops and prevents such procedures from accessing array elements which are not explicitly passed to it.

Nested forall loops can be used to specify multiple levels of parallelism. At this point, Vienna Fortran allows only tightly nested forall loops and provides special syntax to combine the loop control variables into a single header.

Example 23 *Nested Forall Loops*

```
PROCESSORS P(PN, PN)
REAL A(N, M) DIST ( BLOCK, BLOCK)
   ⋮
FORALL (I = 1, N, J = 1, M, 2)
      A(I, J ) = ...
         ⋮
END FORALL
```

The above code segment specifies a doubly nested forall loop of $N*((M+1) \div 2)$ *iterations where I ranges from* 1 *to N while J ranges from* 1 *to M by* 2. □

4.3 Work Distribution

The compiler, by default, assigns loop iterations to processors for execution. The strategy for such distribution of work could be as simple as assigning a block of iterations to a processor. A more efficient approach would result from analyzing the array access patterns of the loop body and then assigning iterations to processors so as to minimize communication while attempting to balance the load. This assignment of work could be at the iteration level, i.e., assign full iterations to a single processor, or if necessary the compiler may choose to break up a single iteration across multiple processors.

Following Kali [26], Vienna Fortran allows users to control the assignment of work 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 a processor element name as shown in the example below:

```
FORALL I = 1, N ON OWNER(A(I))
      ⋮
END FORALL
```

Here the on clause is specified using the intrinsic function, **OWNER**, which returns the home processor of its argument. Thus, the above on clause specifies that the Ith iteration of the forall loop should be executed on the processor which owns the array element $A(I)$. If an array is replicated across several processors, then the intrinsic function **OWNER** returns a system specific result.

Although, the above example is the most common use of the on clause, it is also possible to name the processor directly, as shown below:

```
FORALL I = 1, $NP ON $P(I)
```

In this case, the on clause is specified using the implicit processor array $\$P$. The number of iterations is the same as the number of processors ($\$NP$), with the Ith iteration being assigned to the Ith processor. Thus, the on clause can be used, as shown above, to specify processor specific code.

4.4 Reduction Operators

As indicated before, forall loops in Vienna Fortran are not allowed to have inter-iteration dependencies. One consequence of this restriction is that no scalar variable can be modified in a forall loop. Vienna Fortran allows reduction operators to represent operations such as summation across the iterations of a parallel loop.

The form of the reduction statement is as follows:

REDUCE (*reduction-op* , *variable* , *expression* [, *order*])

It consists of a reduction operator, a target variable and a expression to be accumulated onto the target variable. The effect of the statement is to accumulate the values of the expression arising out of the different loop iterations onto the target variable. The order in which the reduction occurs may be left to the system or as in SISAL [24], it may be explicitly specified to be *LEFT*, *RIGHT* or *TREE* order.

Example 24 Summing values of a distributed array

```
      X = 0.0
      FORALL 10 I = 1, N ON OWNER(A(I))
            ...
         REDUCE( ADD , X, A(I))
            ...
10    CONTINUE
```

In this loop, the reduction statement along with the reduction operator **ADD** *is used to sum the values of the distributed array* A *onto the variable* X.□

Along with **ADD**, Vienna Fortran provides **MULT**, **MAX**, and **MIN** as reduction operators. A user defined function can be used as a reduction operator as follows:

```
REDUCE( func , X, A(I))
```

where $func$ is a user defined function with two input arguments. The semantics of the above statement are as follows: the function $func$ is called with X and $A(I)$ as arguments and the result is assigned as the new value of X. The functions used as reduction operators must be commutative and associative (within the constraints of floating point operations) for consistent results.

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.

5 Specification of Distribution and Alignment Functions

Vienna Fortran provides a facility for the explicit specification of distribution functions and alignment functions, thereby allowing the user to extend the set of intrinsic functions defined in the language. The specification of a distribution function introduces a class of **distribution types** by establishing mappings from (data) arrays to processor arrays, while an alignment function determines a class of mappings from source arrays to target arrays, each of which constitutes an alignment in the sense of Section 2.4.

Distribution functions and alignment functions both are not functions in the ordinary FORTRAN 77 sense in that their activation results in the computation of a distribution or alignment, respectively, rather than in the computation of a (scalar) value. Apart from this, no side effects may occur as a result of executing these functions.

5.1 Specification of Distribution Functions

5.1.1 Syntax

1. dfunction-statement → **DFUNCTION** dfunction-name ["("[dummy-argument-list]")"]
2. dummy-argument → *variable_name* | *array_name* | *procedure_name*
3. target-array-specification → **TARGET** generalized-array-declarator
4. processor-array-specification → **PROCS** generalized-array-declarator
5. distribution-mapping-statement → distribution-index-mapping | distribution-dimension-mapping
6. distribution-index-mapping → data-reference **DIST** ["("distribution-expression")"]
   TO processor-reference
7. distribution-dimension-mapping → array-dimension **DIST** ["("distribution-function-reference")"]
   TO processor-dimension
8. processor-dimension → array-dimension
9. end-dfunction-statement → **END DFUNCTION** [dfunction-name]

5.1.2 Semantics

Distribution functions are specified similar to *function_subprograms*:

DFUNCTION f $[(d_1, \ldots, d_k)]$
specification_statements
executable_statements
END DFUNCTION $[f]$

where

- f is the *function name*
- $(d_1, \ldots, d_k)$ is an optional list of **explicit dummy arguments** ($k \geq 0$). Their specification and use is governed by standard Fortran rules. If no explicit dummy arguments are specified, the parentheses can be omitted.

- In addition to explicit dummy arguments, each distribution function has two **implicit dummy arguments**, which respectively are associated with the array to be distributed (the **target array**), and the processor array which is the target of the distribution.

  We illustrate the association between the implicit dummy arguments of a distribution function f and the corresponding actual arguments by an example: consider the annotated static array declaration

  $$\textbf{REAL } A1(N,M),\ A2(N,N),\ A3(M,N) \textbf{ DIST}\,(f(K)) \textbf{ TO } R2$$

  The evaluation of this declaration results in three successive calls to f:

  $$f(K)\{A1, R2\}$$
  $$f(K)\{A2, R2\}$$
  $$f(K)\{A3, R2\}$$

  Here, the implicit actual arguments are enclosed between set brackets.

- The *specification_statement*s must include a *target-array-specification* for the implicit dummy array argument, A, and a *processor-array-specification* for the implicit dummy processor array argument, R. These two specifications begin with the keyword **TARGET** or **PROCS** , respectively, followed by a *generalized-array-declarator* (see Section 3.1.2). If n and m are permissible ranks of the actual arguments associated with A and R, respectively, then f is designated as an **n-m distribution function**.

  If the rank of both implicit dummy arguments is specified as 1 and a reference to f occurs in a *composite distribution expression*, then the function performs a mapping from one array dimension to one processor array dimension, as specified in Section 3.4.7.

  A *processor-declaration* may or may not be included, following the general rules discussed in sections 3.2 and 3.10.

- The set of *executable_statement*s includes a *distribution-mapping-statement*, which is used to explicitly specify the components of a distribution, either element–wise or dimension–wise. The method used to specify the mapping is discussed in detail below.

  Apart from their use in distribution mapping statements, elements of A may neither be referenced nor defined in the distribution function.

- A reference to a distribution function must explicitly specify for each index of the target array how the corresponding element is to be mapped to a non-empty set of processors. More precisely, for each $\mathbf{i} \in \mathbf{I}^A$, a mapping from $\mathbf{i}$ to a non-empty set of processor indices must be explicitly specified by the application of *distribution mapping statements*. This yields a distribution for A with respect to R.

Distribution Mapping Statements

*Distribution mapping statement*s are used to specify components of the distribution δ_R^A that is to be computed by an activation of the distribution function. Distribution mapping statements fall into two categories – *distribution-index-mapping*s and *distribution-dimension-mapping*s – depending on whether they are element–oriented or dimension–oriented:

- The *distribution-index-mapping* statement has the form:

  dref **DIST** [(*dex*)] **TO** *pref*

where *dref* is a data reference associated with A – i.e., either a single array element or an array section, *dex* is an optional parenthesized distribution expression, consisting only of *distribution function references* and elision symbols, and *pref* is a processor reference – i.e., either a single processor or a processor array section (see Section 3.1). Let $\mathbf{I}$ denote the set of index values associated with *dref*, and $\mathbf{J}$ the corresponding set associated with *pref*. Then $\mathbf{I} \subseteq \mathbf{I}^A$, and $\mathbf{J} \subseteq \mathbf{I}^R$.

Assume first that *dex* is not specified. Then the effect of the index mapping is as follows:

$$\forall \mathbf{i} \in \mathbf{I} : \forall \mathbf{j} \in \mathbf{J} : \mathbf{j} \text{ is included in } \delta_R^A(\mathbf{i})$$

If *dex* is specified, then *dref* is mapped to *pref* according to the definitions of the distribution functions referenced, and to the rules specified in Sections 3.4.6 and 3.4.7.

- The *distribution-dimension-mapping* statement is of the form:

$$A.d1 \textbf{ DIST } [(dref)] \textbf{ TO } R.d2$$

where $d1$ and $d2$ are dimension qualifiers for A and R, respectively, and *dref* is a reference to a 1-1 distribution function. In this case, *dref* specifies the distribution of dimension $d1$ of A by mapping it to dimension $d2$ of R.

5.1.3 Examples

Example 25 Specification of a variant of block distributions

This function specifies an alternative to the intrinsic function BLOCK *(see Section 3.4.3). It partitions the target array (dimension) into a number of contiguous intervals of the same size, possibly followed by one interval of a smaller size. There may be one or more processors that do not receive any elements of the array.*

```
      DFUNCTION BLOCK1
      TARGET D(0:)
      PROCS P(0:)
      INTEGER LBL0, LBL1, UBD

      LBL0 = SIZE(D) / SIZE(P)
      LBL1 = SIZE(D) / SIZE(P) + 1
      UBD = MIN( SIZE(P), SIZE(D)) - 1

      IF (LBL0 * SIZE(P) .EQ. SIZE(D)) THEN
      DO 11 J = 0, UBD
         D(J*LBL0 : (J+1)*LBL0 - 1) DIST TO P(J)
11    CONTINUE
      ELSE
      DO 22 J = 0, UBD
         D(J*LBL1 : MIN( UBOUND(D,1), (J+1)*LBL1 - 1) DIST TO P(J)
22    CONTINUE
      END IF

      END DFUNCTION BLOCK1
```

Example 26 Specification of the intrinsic function $CYCLIC(K)$

```
DFUNCTION CYCLIC(K)
TARGET D(0:)
PROCS P(0:)
INTEGER K,N

N= UBOUND(D,1)
DO 11 I = 0, N, K
   D(I : MIN(N, I+K-1)) DIST TO P(MOD(I/K, $NP)
11 CONTINUE

END DFUNCTION CYCLIC
```

Example 27 Specification of the intrinsic function $INDIRECT$

In the distribution function specified below, we assume $\mathbf{I}^C = \mathbf{I}^A$. *A and C may be of an arbitrary rank.*

```
DFUNCTION INDIRECT(C)
TARGET A(*)
PROCS P(:)
INTEGER C(*)

DO 11 I = 1, SIZE(A)
  A(I) DIST TO P(C(I))
11 CONTINUE

END DFUNCTION INDIRECT
```

Example 28 Generating rectangular stripes

An array of rank 1 *is distributed in* k *stripes of arbitrary width. The stripe boundaries are given in the array parameter* l*: for all* $1 \leq j \leq k, l(j)$ *is the lower bound, and* $l(j+1)-1$ *is the upper bound of* $stripe_j$.

```
DFUNCTION stripes (k,l)
TARGET D(:)
PROCS P(0:)
INTEGER k, l(k+1)

DO 11 jj = 1, k
  D(l(jj) : l(jj+1)-1) DIST TO P(MOD(jj, $NP))
11 CONTINUE

END DFUNCTION stripes
```

Example 29 Generating diagonal stripes

```
DFUNCTION dstripes
TARGET D(1:,1:)
PROCS P(0:)
INTEGER UB2

UB2= UBOUND(D,2)
DO I = 1, UBOUND(D,1)
   DO J = 2, UB2
      D(I,J) DIST TO P(MOD(I - J + UB2, $NP)
   END DO
END DO

END DFUNCTION dstripes
```

Example 30 Replication of stripes

The k*th stripe – which is of length* $l(k)$ *–is mapped to the processor index range* $pn(k,1) : pn(k,2)$.

```
DFUNCTION repstripes (k,l,pn)
TARGET D(:)
PROCS P(:)
INTEGER k, l(k), pn(k,2), lower, upper

lower = LBOUND(D,1)
DO jj = 1, k
   upper = lower + l(jj)
   D(lower : upper-1) DIST TO P(pn(jj,1) : pn(jj,2))
   lower = upper
END DO
END DFUNCTION repstripes
```

Example 31 Cyclic distribution of two dimensions

```
DFUNCTION bicycle (M,N)
TARGET D(:,:)
PROCS P(:,:)

D.1 DIST CYCLIC(M) TO P.1
D.2 DIST CYCLIC(N) TO P.2

END DFUNCTION bicycle
```

□

5.2 Specification of Alignment Functions

5.2.1 Syntax

1. afunction-statement → **AFUNCTION** afunction-name ["("[dummy-argument-list]")"]
2. source-array-specification → **SOURCE** generalized-array-declarator
3. alignment-mapping-statement → data-reference **ALIGN** ["("alignment-function-reference")"] **WITH** data-reference
4. end-afunction-statement → **END AFUNCTION** [afunction-name]

5.2.2 Semantics

Alignment function specifications specify a class of mapping from a set of **target arrays** to a set of **source arrays**. Virtually all that has been said about distribution functions can be carried over to the discussion of alignment functions, if the role of the processor reference is replaced by the source array data reference. Therefore we keep this discussion to a minimum length.
Alignment functions are specified as follows:

AFUNCTION f $[(d_1, \ldots, d_k)]$
specification_statements
executable_statements
END AFUNCTION $[f]$

where the notation has the analogous meaning as for distribution functions.

The difference between distribution and alignment functions is as follows: while the two implicit arguments of a distribution function specify the target array to be distributed and the processor array, the two implicit dummy arguments of an alignment function are the **target array**, A, and the **source array**, B, of an alignment. An activation of f yields an alignment $\alpha : \mathbf{I}^A \rightarrow \mathbf{I}^B$.

The specifications of the target and source arrays take respectively the form

TARGET *generalized-array-declarator*

and

SOURCE *generalized-array-declarator*

For both specifications, the same rules hold as discussed in Section 3.1.2.

The set of *executable_statements* includes the *alignment-mapping-statement*. In contrast to the *distribution-mapping-statement*, there is only one version of this, which specifies index mapping:

tref **ALIGN** [(*afr*)] **WITH** *sref*

where

- *tref* represent a target array reference
- *afr* is an optional alignment function reference
- *sref* is the source array reference

The syntax and semantics of this construct are analogous to that specified for the *distribution-index-mapping*, with *tref* substituted for the data reference *dref*, *afr* playing the role of the distribution expression, *dex*, and *sref* substituted for the processor reference *pref*.

Apart from their use in mapping statements, elements of A and B may neither be referenced nor defined in the function.

Assume first that *afr* is not specified. Let $\mathbf{I}$ denote the set of index values associated with *tref*, and $\mathbf{J}$ the corresponding set associated with *bref*. Then $\mathbf{I} \subseteq \mathbf{I}^A$, and $\mathbf{J} \subseteq \mathbf{I}^B$. The effect of the alignment mapping statement is then as follows:

$$\forall \mathbf{i} \in \mathbf{I} : \forall \mathbf{j} \in \mathbf{J} : \mathbf{j} \text{ is included in } \alpha(\mathbf{i})$$

If an alignment function reference is specified, a mapping is performed as specified by *tref, sref*, and *afr*. See Section 3.5.4 for details.

Example 32 Transposition

```
      AFUNCTION transpose
      TARGET T(1:,1:)
      SOURCE B(1:,1:)

      DO 11 I = 1, UBOUND(T,1)
         DO 11 J = 1, UBOUND(T,2)
            T(I,J) ALIGN WITH B(J,I)
11    CONTINUE

      END AFUNCTION transpose
```

Example 33 Skewed alignment

The main diagonal of the target array is aligned with the second dimension of the source array.

```
      AFUNCTION skewed
      TARGET T(1:,1:)
      SOURCE B(1:,1:)
      DO 11 I = 1, UBOUND(T,1)
      DO 11 J = 1, UBOUND(T,2)
            T(I,J) ALIGN WITH B(I, UBOUND(B,2)+1-J)
11    CONTINUE
      END AFUNCTION skewed
```

□

6 Concurrent Input/Output Statements

6.1 Syntax

1. concurrent-io-statement → copen-statement | cclose-statement | cwrite-statement | cread-statement | cbackarray-statement | cskip-statement | crewind-statement

2. copen-statement → **COPEN** "(" copenitem-list ")"

3. copenitem → [**UNIT** "="] external-unit-identifier | **IOSTAT** "=" *integer_variable* | **ERR** "=" *label* | **FILE** "=" (*name* | *character_string*) | **STATUS** "=" ('**OLD**' | '**NEW**') | **FORM** "=" ('**FORMATTED**' | '**UNFORMATTED**')

4. cclose-statement → **CCLOSE** "(" ccloseitem-list ")"

5. ccloseitem → [**UNIT** "="] external-unit-identifier | **IOSTAT** "=" *integer_variable* | **ERR** "=" *label* | **STATUS** "=" ('**KEEP**' | '**DELETE**')

6. cwrite-statement → **CWRITE** "(" cwriteitem-list ")" *array_name*-list

7. cwriteitem → [**UNIT** "="] external-unit-identifier | [**EXTDIST** "="] **SYSTEM** | **IOSTAT** "=" *integer_variable* | **ERR** "=" *label*

8. external-unit-identifier → *integer_expr*

9. cread-statement → **CREAD** "(" creaditem-list ")" *array_name*-list

10. creaditem → [**UNIT** "="] external-unit-identifier | **END** "=" *label* | **IOSTAT** "=" *integer_variable* | **ERR** "=" *label*

11. cbackarray-statement → **CBACKARRAY** "(" cbackarrayarrayitem-list ")"

12. cbackarrayitem → [**UNIT** "="] external-unit-identifier | **IOSTAT** "=" *integer_variable* | **ERR** "=" *label*

13. cskip-statement → **CSKIP** "(" cskipitem-list ")"

14. cskipitem → [**UNIT** "="] external-unit-identifier | ([**ARRN** "="] ("*" | *integer_expression*) | **IOSTAT** "=" *integer_variable* | **ERR** "=" *label*)

15. crewind-statement → **CREWIND** "(" crevitem-list ")"

16. revlist-item → [**UNIT** "="] external-unit-identifier | **IOSTAT** "=" *integer_variable* | **ERR** "=" *label*

6.2 Semantics

In this section, we describe the file operations provided by Vienna Fortran for input/output of distributed data structures to a concurrent file system These operations are in addition to the usual FORTRAN 77 file operations which can be utilized for input and output of scalar and replicated data.

We presume that the underlying machine supports a concurrent file system, e.g., CFS on the Intel parallel machines, which allows files to be stored and accessed in parallel. The system may include a set of I/O nodes each of which controls one or more disks. The I/O nodes are part of the communication subsystem of the machine which supports simultaneous parallel access to the disk drives by the processors. We presume that the actual mapping of a file to physical blocks on the set of disks is handled by the communication subsystem, for example, by striping the file across the disks.

The communication subsystem should allow the same concurrent file to be opened and read by a number of different processes where each process maintains its own file pointer and is not affected by any of the other processes. This allows each process to be responsible for the input/output of its local segment of a distributed data structure.

A Vienna Fortran concurrent file has a special structure and can be opened and accessed only through the operations described below. Such a file consists of a sequence of records; each record contains an array distribution descriptor followed by data elements of the array. The distribution descriptor contains enough information to regenerate the data distribution for the array.

The Vienna Fortran concurrent I/O statements described below are considered to be executed synchronously. That is, all processors execute each I/O statement and (conceptually) no processor can proceed until all the processors have completed the execution of the I/O statement.

The simple form of the statement for opening a Vienna Fortran concurrent file is as follows:

COPEN (**UNIT** = *integer_expr*, **FILE** = *character_string*, **STATUS** = *sta*)

This is the same as the FORTRAN 77 open statement except we restrict the set of possible specifiers. **COPEN** signals an *error* state if the file with the given name exists but it has been created by a standard FORTRAN 77 open statement.

A **CCLOSE** statement is used to terminate the connection of a particular file to a unit. The simple form of this statement is as follows:

CCLOSE (**UNIT** = *integer_expr*, **STATUS** = *sta*)

In this statement we allow the same set of specifiers as provided by the FORTRAN 77 close statement.

One or more distributed arrays can be written out to a Vienna Fortran concurrent file by using the **CWRITE** statement. There are two forms of the **CWRITE** statement. The first one is as follows:

CWRITE (**UNIT** = *integer_expr*) ad_1, ad_2, ..., ad_r

where ad_i, $1 \leq i \leq r$ are array identifiers. The file record written for each distributed array, ad_i, consists of the distribution descriptor followed by the sequence of all the data elements of ad_i. This sequence results from the concatenation of linearized local segments of ad_i owned by the individual processors. These segments are concatenated according to the increasing order of logical numbers of the processors that own the corresponding segments. The logical number of a processor is its index in the default processor array $\$P$.

The distribution descriptor stores information describing: 1) the distribution type of ad_i, 2) the index set of ad_i, and 3) the index set of the processor array to which the array, ad_i, is distributed.

The above form of the **CWRITE** statement is useful and most efficient when the array being written is going to be read back into a target array with exactly the same distribution. In such a situation, the local segments of the array structure can be read directly in parallel by all the processors involved. When these records are read into target arrays whose distribution does not match the distribution used for writing out the elements, a redistribution through distributed temporary arrays is necessary. If it is known that a distributed array being output is to be read into an array with a different distribution, we can use the second form of the **CWRITE** statement. As shown below, this form specifies that the external distribution used for writing out the arrays should be a a standard *system* default distribution:

CWRITE (**UNIT** = *integer_expr*, **EXTDIST = SYSTEM**) ad_1, ad_2, ..., ad_r

The default *system* distribution used by Vienna Fortran, is one in which only the last dimension of an array is distributed by the *BLOCK* distribution. Using this default distribution, the array elements will always be written in a sequence that corresponds to the standard FORTRAN 77 array element ordering, independent of the number of processors executing the statement.

The reading of one or more distributed arrays is specified by a statement of the following form:

CREAD (**UNIT** = *integer_expr*) ad_1, ad_2, ..., ad_r

where ad_i, $1 \leq i \leq r$ are again array identifiers. The distribution information is read from the file and the compatibility with the distributions of ad_i is checked. If necessary, a redistribution through a distributed temporary array is performed.

There are some restrictions on the use of **CREAD** and **CWRITE** statements:

- No mixing of **CREAD** and **CWRITE** statements is allowed on an open file.
- **CREAD** and **CWRITE** statements can only be applied to files opened by a **COPEN** statement.

As noted above, parallel access to the distributed files requires each processor executing the Vienna Fortran program to maintain independent file pointers to the same file. However, conceptually there is a single logical file pointer which allows all the processors to be working on the same record in the file. The following operations allow for positioning of the logical file pointer within a distributed file. Given the SPMD nature of the generated code, each processor executes the same statement, thus maintaining a consistent position in the file.

Execution of a **CBACKARRAY** statement causes the file to be positioned before the preceding record. The simple form of this statement is as follows:

CBACKARRAY (**UNIT** = *integer_expr*)

The statement **CSKIP** serves either for skipping to the end of file:

CSKIP (**UNIT** = *integer_expr*, **ARRN** = *)

or for skipping forward over records:

CSKIP (**UNIT** = *integer_expr*, **ARRN** = *integer_expression*)

The number of records skipped is specified by the *integer_expression*.

Execution of a **CREWIND** statement

CREWIND (**UNIT** = *integer_expr*)

causes the specified file to be positioned at its initial point.

Example 34 Writing and reading arrays.

```
      ASSERT  R.GE.2
      PROCESSORS P2D(R,R) RESHAPE P1D(R*R)

      REAL A(N, N) DIST ( BLOCK, BLOCK )
      REAL B(N, N) DIST ( BLOCK, BLOCK )
      REAL V(N*N)  DIST ( BLOCK ) TO P1D

C ... initialization of A by some algorithm ...
      COPEN( UNIT = 7, FILE = '/usr/example1',STATUS = 'NEW')
      CWRITE (7) A
      CCLOSE (7)

      ...
```

COPEN (**UNIT** = 7, **FILE** = '/usr/example1', **STATUS** = 'OLD')
CREAD (7) B
CBACKARRAY (7)

CREAD (7) V
CCLOSE (7)

If, e.g. $R = 2$ and $N = 4$, the array A is distributed in the following way.

```
Segment1     Segment3

 a11  a12 | a13  a14              Segment1 is owned by P2D(1,1)
          |
 a21  a22 | a23  a24              Segment2 is owned by P2D(2,1)
 -------------------
 a31  a32 | a33  a34              Segment3 is owned by P2D(1,2)
          |
 a41  a42 | a43  a44              Segment4 is owned by P2D(2,2)

Segment2     Segment4
```

The segments are written to the file /usr/example1 *in the following order:* Linearized Segment1, Linearized Segment2, Linearized Segment3, Linearized Segment4. *Thus, data elements of* A *appear in the corresponding file record in the following order:* a11 a21 a12 a22 a31 a41 a32 a42 a13 a23 a14 a24 a33 a43 a34 a44. *Since the array* B *has the same distribution as the array* A, *the data elements of* A *are transferred to* B *in order, i.e., b_{ij} is the same as a_{ij}. However, since the array* V *is one-dimensional, the above elements of* A *are transferred to the elements* v1, v2, ..., v16 *of the array* V *respectively.* □.

Acknowledgments

The work contained in this document has been the subject of a large number of discussions and debates over the past year; it is impossible to mention all those researchers and applications programmers who have contributed in some way to the features of this language or who have helped us to understand some of the issues involved. We would, however, particularly like to thank Marina Chen, Tom Eidson, Mark Furtney, Michael Gerndt, Irene Qualters, Joel Saltz, John Van Rosendale and the Fortran D group at Rice University for their helpful comments and discussions. Last, but not least, we would like to thank all the other members of our research group at the University of Vienna.

References

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19:26–34, August 1986.

[2] F. André, J.-L. Pazat, and H. Thomas. PANDORE: A system to manage data distribution. In *International Conference on Supercomputing*, pages 380–388, June 1990.

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

[4] S. Benkner, B. Chapman, and H. Zima. Vienna Fortran 90. In *Proceedings of the SHPCC Conference 1992 (to appear)*, 1992.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] A. L. Cheung and A. P. Reeves. The Paragon multicomputer environment: A first implementation. Technical Report EE-CEG-89-9, Cornell University, Ithaca, NY, July 1989.

[11] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[12] 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.

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

[14] P. Hatcher, A. Lapadula, R. Jones, M. Quinn, and J. Anderson. A production quality C* compiler for hypercube machines. In *3rd ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, pages 73–82, April 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] 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.

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

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

[24] J. McGraw, S. Skedzielewski, S. Allan, R. Oldenhoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language: Language reference manual. Report M-146, Lawrence Livermore National Laboratory, March 1985.

[25] 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.

[26] 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.

[27] *MIMDizer User's Guide, Version 7.02.* Pacific Sierra Research Corporation, Placerville, CA., 1991.

[28] 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.

[29] 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.

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

[31] D. Pountain. *A Tutorial Introduction to Occam Programming.* Inmos, Colorado Springs, Co., 1986.

[32] A. P. Reeves. Paragon: a programming paradigm for multicomputer systems. Technical Report EE-CEG-89-3, Cornell University, January 1989.

[33] 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.

[34] M. Rosing, R. W. Schnabel, and R. P. Weaver. Expressing complex parallel algorithms in DINO. In *Proceedings of the 4th Conference on Hypercubes, Concurrent Computers, and Applications*, pages 553–560, 1989.

[35] M. Rosing, R. W. Schnabel, and R. P. Weaver. The DINO parallel programming language. Technical Report CU-CS-457-90, University of Colorado, Boulder, CO, April 1990.

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

[37] 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.

[38] *CM Fortran Reference Manual, Version 5.2.* Thinking Machines Corporation, Cambridge, MA, 1989.

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

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

[41] M. Wu and G. Fox. Fortran 90D Compiler for distributed memory MIMD parallel computers. Technical Report SCCS-88b, Syracuse University, 1991.

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

[43] 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.

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

A Examples

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.

A.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 1, 2, and 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 two-dimensional processor array and a cyclic distribution in both dimensions of A may be superior in some cases (cf. [40])).

Hence, as shown in Figure 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 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 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 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,

```
      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 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) = 0
        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 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 3: Subroutine for the solution step of Gaussian Elimination

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.

A.2 ADI Iteration

ADI (Alternating Direction Implicit) is a well known and effective method for solving partial differential equations in two or more dimensions [23]. 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 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**.

```
      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 4: An ADI iteration

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.

A.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 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, 36] 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 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 5: Relaxation sweep over an unstructured mesh

B Intrinsic Functions

This section describes the intrinsic functions (including intrinsic distribution functions) of Vienna Fortran. We will use the following notation:

- *alloc-array* allocatable array
- *array* arbitrary array
- *dim* integer constant
- *element* data element (scalar or array element)
- *int-array* integer array with index domain $[1 : N]$ for some N.
- *len* integer constant ≥ 1
- *mask* logical array
- *processor* processor element or processor index (relating to $\$P$)
- *query* query without name tag in the sense of Section 3.8.3

B.1 ALL

Form: *ALL(mask)*

Result type: logical
Result value: Determines whether all values are true in *mask*.

B.2 ALLOCATED

Form: *ALLOCATED(alloc-array)*

Result type: logical
Result value: Determines whether the allocation status of *alloc-array* is **allocated**.

B.3 ANY

Form: *ANY(mask)*

Result type: logical
Result value: Determines whether any value is true in *mask*.

B.4 BLOCK

Form: *BLOCK*

Result type: 1-1 distribution function
Result: Creates a block distribution. See Section 3.4.3.

B.5 B_BLOCK

Form: *B_BLOCK(int-array)*

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*(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.

B.6 CYCLIC

Form: *CYCLIC([len])*

Result type: 1-1 distribution function
Default: The default value for *len* is 1
Result: Depending on whether or not $len = 1$, a cyclic or a block cyclic distribution is generated. See Section 3.4.3.

B.7 CYCLIC_LEN

Form: *CYCLIC_LEN(array,dim)*

Result type: integer
Result value: The distribution type associated with the *distribution extraction* (= *array.dim*) must be *CYCLIC(K)*. The function determines the block length, K.

B.8 DISTRIBUTED

Form: *DISTRIBUTED(array)*

Result type: logical
Result value: Determines whether the argument array has a defined distribution.

B.9 DYNAMIC

Form: *DYNAMIC(array)*

Result type: logical
Result value: Determines whether the argument array is a dynamically distributed array.

B.10 IDT

Form: *IDT(array[,dim],query)*

Result type: logical
Result: Let t denote the type associated with the *distribution extraction* (= $A.dim$) or (= A), depending on whether or not *dim* is specified (see Section 3.4.4). The reference to *IDT* yields **true** iff t and *query* match according to the rules specified in Section 3.8.3.

B.11 IDTA

Form: $IDTA(array_1[,dim_1],array_2[,dim_2])$

Result type: logical
Result: The effect of $IDTA(array_1[,dim_1],array_2[,dim_2])$ is equivalent to the effect of $IDT(array_1[,dim_1],(=array_2[.dim_2]))$.

B.12 INDIRECT

Form: *INDIRECT(array)*

Result type: distribution function
Result: Creates an indirect distribution. See Section 3.4.3.

B.13 LBOUND

Form: *LBOUND(array,dim[,processor])*

Result type: integer
Result value:

- If *processor* is not specified, then the function returns the lower bound of *array* in dimension *dim*.
- If *processor* is specified, then the function return the smallest i such that $array(\ldots,i,\ldots)$, where i occurs in position *dim*, is owned by *processor*.

B.14 OWNED

Form: *OWNED(element,processor)*

Result type: logical
Result value: Determines whether *element* is owned by *processor*.

B.15 OWNER

Form: *OWNER(element)*

Result type: integer
Result value: Determines i such that $\$P(i)$ owns *element*. If i is not uniquely determined, a system-dependent index is returned.

B.16 SIZE

Form: *SIZE(array[,dim][,processor])*

Result type: integer
Default: If *dim* is specified, then the *operand* of the function is the corresponding dimension of *array*. If *dim* is not specified, then the *operand* is the whole *array*.
Result value:

- If *processor* is not specified, then the function returns the number of elements in the operand.

- If *processor* is specified, then the function returns the number of elements of the operand owned by *processor*.

B.17 S_BLOCK

Form: *S_BLOCK(int-array)*

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*(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.

B.18 UBOUND

Form: *UBOUND(array[,dim][,processor]*

Result type: integer
Result value:

- If *processor* is not specified, then the function returns the upper bound of *array* in dimension *dim*.
- If *processor* is specified, then the function return the largest i such that *array*$(\ldots, i, \ldots)$, where i occurs in position *dim*, is owned by *processor*.

B.19 $MY_PROC

Form: *$MY_PROC*

Result type: integer
Result value: Determines the index of the executing processor in $\$P$.

B.20 $NP

Form: *$NP*

Result type: integer
Result value: Determines the number of processors executing the program.

C 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 Basic Elements

1. assertion → **ASSERT** "(" *expression* ")"
2. array-section → *array_name* ["("section-subscript-list")"]
3. section-subscript → subscript | subscript-triplet
4. subscript → *integer_expr*
5. subscript-triplet → [subscript] ":" [subscript] [":" stride]
6. stride → *integer_expr*
7. data-reference → *array_element_name* | array-section
8. generalized-array-declarator → *array_declarator* | assumed-shape-array-declarator
9. assumed-shape-array-declarator → *array_name* "(" assumed-shape-spec-list ")"
10. assumed-shape-spec → [*dim_bound_expr*] ":"
11. declaration-annotation → actual-array-annotation | dummy-array-annotation
12. actual-array-annotation → static-array-annotation | dynamic-array-annotation
13. extension-executable-statement → distribute-statement | allocate-statement | deallocate-statement | forall-loop | dcase-construct | concurrent-io-statement

C.3 Processor Declarations

1. processor-declaration → primary-processor-structure [secondary-processor-structures]
2. primary-processor-structure → **PROCESSORS** generalized-array-declarator
3. secondary-processor-structures → **RESHAPE** generalized-array-declarator–list

C.4 Processor References

1. processor-reference → processor-element-name | processor-section ["(/"dimension-permutation"/)"]
2. processor-element-name → *array_element-name*
3. processor-section → array-section
4. dimension-permutation → *int_constant_expr*–list

C.5 Distribution Expressions

1. distribution-expression → simple-distribution-expression | composite-distribution-expression
2. simple-distribution-expression → distribution-function-reference | distribution-extraction | distribution-type-name
3. distribution-function-reference → *function_reference*
4. distribution-extraction → "=" array-or-dimension
5. array-or-dimension → *array_name* | array-dimension
6. array-dimension → *array_name* dimension-qualifier
7. dimension-qualifier → "." *int_constant_expr*
8. distribution-type-definition → **DTYPE** "(" dtype-pair-list ")"
9. dtype-pair → name "=" "("distribution-expression")"
10. composite-distribution-expression → dimensional-expression-list
11. dimensional-expression → simple-distribution-expression | ":"

C.6 Alignment Specifications

1. alignment-specification → **ALIGN** aspec
2. aspec → alignment-expression | functional-alignment
3. alignment-expression → target-array-identification "("bound-variable-list")" **WITH** source-array-reference
4. target-array-identification → *array_name* | "$"
5. bound-variable → *variable_name* | ":"
6. source-array-reference → data-reference
7. functional-alignment → "(" alignment-function-reference ")" **WITH** source-array-section
8. alignment-function-reference → *function_reference*

C.7 Static Array Annotations

1. static-array-annotation → [distribution-specification | alignment-specification]
2. distribution-specification → **DIST** dspec
3. dspec → "("distribution-expression")" [**TO** processor-reference] | **TO** processor-reference

C.8 Dynamically Distributed Arrays

1. dynamic-array-annotation → **DYNAMIC** (primary-array-annotation | secondary-array-annotation)
2. primary-array-annotation → ["," distribution-range] ["," initial-distribution]
3. distribution-range → **RANGE** "("dspec-list")"
4. initial-distribution → distribution-specification | alignment-specification
5. secondary-array-annotation → ","**CONNECT** connection
6. connection → distribution-extraction | aspec
7. distribute-statement → **DISTRIBUTE** distribution-group
8. distribution-group → *array_name*–list "::" [(dspec | alignment-specification)] [notransfer-attribute]
9. notransfer-attribute → **NOTRANSFER**["("*array_name*–list")"]

C.9 Control Constructs

1. control-construct → dcase-construct | if-construct
2. dcase-construct → select-dcase-statement condition-action-pair... end-select-statement
3. select-dcase-statement → **SELECT DCASE** "(" *array_name*–list ")"
4. condition-action-pair → **CASE** condition action
5. condition → query-list | **DEFAULT**
6. query → [name-tag] (dspec | "*")
7. name-tag → *array_name* ":"
8. action → [*executable_statement*]...
9. end-select-statement → **END SELECT**
10. if-construct → *logical_if_statement* | *block_if_statement* | *else_if_statement*

C.10 Allocatable Arrays

1. allocatable-array-declarator → *array_name* "(" deferred-shape-spec-list ")" allocatable-attribute
2. deferred-shape-spec → ":"
3. allocatable-attribute → **ALLOCATABLE**
4. allocatable-array-annotation → actual-array-annotation
5. allocate-statement → **ALLOCATE** "("allocation-list")"
6. allocation → *array_declarator*
7. deallocate-statement → **DEALLOCATE** "("*array_name*–list")"

C.11 Procedures

1. dummy-array-annotation → actual-array-annotation [dummy-annotation-attribute]... | inherit-annotation
2. inherit-annotation → **DIST** "(" "*" ")" ["," distribution-range]
3. dummy-annotation-attribute → restore-attribute | nocopy-attribute | notransfer-attribute
4. restore-attribute → **RESTORE**
5. nocopy-attribute → **NOCOPY**

C.12 FORALL Loops

1. forall-loop → forall-statement private-var-decls forall-block end-forall
2. forall-statement → label-forall-statement | nonlabel-forall-statement
3. label-forall-statement → [forall-construct-name ":"] **FORALL** *label* forall-control
4. nonlabel-forall-statement → [forall-construct-name ":"] **FORALL** forall-control
5. forall-control → (control-variable | "(" control-variable-list ")") [on-clause]
6. control-variable → *variable_name* "=" *integer_expr* "," *integer_expr* ["," *integer_expr*]
7. on-clause → **ON** processor-element-name
8. processor-reference → **OWNER** "(" *array_element_name* ")" | processor-element-name
9. private-var-decls → *dimension_statement* | *type_statement*
10. forall-block → allocate-statement | deallocate-statement | reduction-statement | *executable_statement*
11. end-forall → end-forall-statement | *continue_statement*
12. end-forall-statement → **END FORALL** [forall-construct-name]
13. reduction-statement → **REDUCE** "(" reduction-op "," *variable* "," *expression* ["," order] ")"
14. reduction-op → **SUM** | **MULT** | **MAX** | **MIN** | *function_name*
15. order → **LEFT** | **RIGHT** | **TREE**

C.13 Specification of Distribution Functions

1. dfunction-statement → **DFUNCTION** dfunction-name ["("[dummy-argument-list]")"]
2. dummy-argument → *variable_name* | *array_name* | *procedure_name*
3. target-array-specification → **TARGET** generalized-array-declarator
4. processor-specification → [PROCS] generalized-array-declarator
5. distribution-mapping-statement → distribution-index-mapping | distribution-dimension-mapping
6. distribution-index-mapping → data-reference **DIST** ["("distribution-expression")"] **TO** processor-reference
7. distribution-dimension-mapping → array-dimension **DIST** ["("distribution-function-reference")"] **TO** processor-dimension
8. processor-dimension → array-dimension
9. end-dfunction-statement → **END DFUNCTION** [dfunction-name]

C.14 Specification of Alignment Functions

1. afunction-statement → **AFUNCTION** afunction-name ["("[dummy-argument-list]")"]
2. source-array-specification → **SOURCE** generalized-array-declarator
3. alignment-mapping-statement → data-reference **ALIGN** ["("alignment-function-reference")"] **WITH** data-reference
4. end-afunction-statement → **END AFUNCTION** [afunction-name]

C.15 Concurrent Input/Output Statements

1. concurrent-io-statement → copen-statement | cclose-statement | cwrite-statement | cread-statement | cbackarray-statement | cskip-statement | crewind-statement
2. copen-statement → **COPEN** "(" copenitem-list ")"
3. copenitem → [**UNIT** "="] external-unit-identifier | **IOSTAT** "=" *integer_variable* | **ERR** "=" *label* | **FILE** "=" (*name* | *character_string*) | **STATUS** "=" ('**OLD**' | '**NEW**') | **FORM** "=" ('**FORMATTED**' | '**UNFORMATTED**')
4. cclose-statement → **CCLOSE** "(" ccloseitem-list ")"
5. ccloseitem → [**UNIT** "="] external-unit-identifier | **IOSTAT** "=" *integer_variable* | **ERR** "=" *label* | **STATUS** "=" ('**KEEP**' | '**DELETE**')
6. cwrite-statement → **CWRITE** "(" cwriteitem-list ")" *array_name*-list
7. cwriteitem → [**UNIT** "="] external-unit-identifier | [**EXTDIST** "="] **SYSTEM** | **IOSTAT** "=" *integer_variable* | **ERR** "=" *label*
8. external-unit-identifier → *integer_expr*
9. cread-statement → **CREAD** "(" creaditem-list ")" *array_name*-list

10. creaditem → [**UNIT** "="] external-unit-identifier | **END** "=" *label* | **IOSTAT** "=" *integer_variable* | **ERR** "=" *label*

11. cbackarray-statement → **CBACKARRAY** "(" cbackarrayarrayitem-list ")"

12. cbackarrayitem → [**UNIT** "="] external-unit-identifier | **IOSTAT** "=" *integer_variable* | **ERR** "=" *label*

13. cskip-statement → **CSKIP** "(" cskipitem-list ")"

14. cskipitem → [**UNIT** "="] external-unit-identifier | ([**ARRN** "="] ("*" | *integer_expression*) | **IOSTAT** "=" *integer_variable* | **ERR** "=" *label*)

15. crewind-statement → **CREWIND** "(" crevitem-list ")"

16. revlist-item → [**UNIT** "="] external-unit-identifier | **IOSTAT** "=" *integer_variable* | **ERR** "=" *label*