Programming in Vienna Fortran*

BARBARA CHAPMAN[1], PIYUSH MEHROTRA[2], AND HANS ZIMA[1]
[1]*Department of Statistics and Computer Science, University of Vienna, Brünner Strasse 72, A-1210 Vienna, Austria*
[2]*ICASE, MS 132C, NASA Langley Research Center, Hampton, VA. 23665*

ABSTRACT

Exploiting the full performance potential of distributed memory machines requires a careful distribution of data across the processors. Vienna Fortran is a language extension of Fortran which provides the user with a wide range of facilities for such mapping of data structures. In contrast to current programming practice, programs in Vienna Fortran are written using global data references. Thus, the user has the advantages of a shared memory programming paradigm while explicitly controlling the data distribution. In this paper, we present the language features of Vienna Fortran for FORTRAN 77, together with examples illustrating the use of these features. © 1992 by John Wiley & Sons, Inc.

1 INTRODUCTION

The continued demand for increased computing power during the last decade has led to the development of computing systems in which a large number of processors are connected, so that their execution capabilities may be combined to solve a single problem.

Several distributed memory processing systems (such as Intel's hypercube series and the nCUBE) have come onto the market and are slowly gaining user acceptance. Other systems are under development or have been announced in recent months. These architectures are relatively inexpensive to build, and are potentially scalable to very large numbers of processors. Hence their share of the market is likely to increase in the near future.

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

A major difficulty with the current generation of distributed memory computing systems is that they generally lack programming tools for software development at a suitably high level. The user is forced to deal with all aspects of the distribution of data and work to the processors, and must control

* The 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.

Received March 1992

Scientific Programming, Vol. 1, pp. 31–50 (1992)
CCC 1058-9244/92/010031-20$04.00

the program's execution at a very low level. This results in a programming style which can be likened to assembly programming on a sequential machine. It is tedious, time-consuming, and error prone. This has led to particularly slow software development cycles and, in consequence, high costs for software.

Thus much research activity is now concentrated on providing suitable programming tools for these architectures. One focus is on the provision of appropriate high-level language constructs to enable users to design programs in much the same way as they are accustomed to on a sequential machine. Several proposals (including ours) have been put forth in recent months for a set of language extensions to achieve this [1–5], in particular (but not only) for Fortran, and current compiler research is aimed at implementing them.

Research in compiler technology has so far resulted in the development of a number of prototype systems, such as Kali [6], SUPERB [7, 8], and the MIMDizer [9]. In contrast to the current programming paradigm, these systems enable users to write code using global data references, as on a shared memory machine, but require them to specify the distribution of the program's data. This data distribution is then used to guide the process of restructuring the code into a single program multiple data (SPMD) program for execution on the target distributed memory multiprocessor. The compiler analyzes the source code, translating global data references into local and nonlocal data references based on the distributions specified by the user. The nonlocal references are satisfied by inserting appropriate message-passing statements in the generated code. Finally, the communication is optimized where possible, in particular by combining messages and by sending data at the earliest possible point in time.

In this paper, we present a machine-independent language extension to FORTRAN 77, Vienna Fortran, which allows the user to write programs for distributed memory multiprocessor systems using global addresses. The Vienna Fortran language extension to Fortran 90 is described in a separate paper [10]. Since the performance of an SPMD program is profoundly influenced by the distribution of its data, most of the extensions proposed here are geared towards allowing the user to explicitly control such distribution of data. Vienna Fortran provides the flexibility and expressiveness needed to permit the specification of parallel algorithms and to carry out the complex task of optimization. Despite this fact, there are relatively few language extensions. A simple algorithm can be parallelized by the addition of just a few constructs which distribute the program's data across the machines.

This paper is organized as follows. In the next section we describe current programming practice on distributed memory MIMD architectures by means of a simple example. Then the programming model assumed by Vienna Fortran is introduced and an overview of the language elements is provided. This paper does not attempt to give a systematic introduction to the whole language, but rather describes some of the most important features by way of simple example codes. These form the body of the subsequent section. Section 5 outlines two more complex problems relevant to real applications, discusses the features of Vienna Fortran which may be used to implement them, and briefly discusses some important features of Fortran programs and how we handle them. Finally, we conclude with a discussion of related work and the implementation status of the Vienna Fortran Compilation System.

2 PROGRAMMING DISTRIBUTED MEMORY SYSTEMS: THE STATE OF THE ART

The current generation of distributed memory multiprocessors is particularly difficult to program: the time taken to adapt existing sequential codes and to develop new applications is prohibitive in comparison to conventional machines, including vector supercomputers. Further, the low level at which programs must be written is the source of both frequent errors and of particularly inflexible codes. Consider the brief example described below.

The Jacobi iterative procedure may be used to approximate the solution of a partial differential equation discretized on a grid. At each step, it updates the current approximation at a grid point by computing a weighted average of the values at the neighboring grid points. An excerpt from a Jacobi relaxation code for execution on a sequential machine is shown in Figure 1.

When this code is parallelized by hand, the programmer must distribute the program's work and data to the processors which will execute it. One of the common approaches to do so makes use of the regularity of most numerical computations. This is the so-called SPMD or data parallel model of computation. With this method, the data arrays in the original program are each partitioned

```
C SEQUENTIAL CODE

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

      CALL INIT (U, F, N)

        ...

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

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

40    CONTINUE

        ...
```

FIGURE 1 Sequential Jacobi relaxation code.

and mapped to the processors. This is known as *distributing* the arrays. The specification of the mapping of the elements of the arrays to the set of processors is called the *data distribution* of that program. A processor is then thought of as *owning* the data assigned to it; these data elements are stored in its local memory. Now the work is distributed according to the data distribution: computations which define the data elements owned by a processor are performed by it—this is sometimes known as the *owner computes* paradigm. The processors then execute essentially the same code in parallel, each on the data stored locally.

It is, however, unlikely that the code on one processor will run entirely without requiring data which is stored on another processor. Accesses to nonlocal data must be explicitly handled by the programmer, who has to insert communication constructs to send and receive data at the appropriate positions in the code. This is called *message passing*. The details of message passing can become surprisingly complex: buffers must be set up, and the programmer must take care to send data as early as possible, and in economical sizes. Several issues arise which do not have their counterpart in sequential programming. New types of errors, such as deadlock and livelock, must be avoided. The programmer must decide when it is advantageous to replicate computations across processors, rather than send data. Moreover, for code which is explicitly parallel, debugging is a serious problem.

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

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

To illustrate this, we reproduce in Figure 2 the above section of code, rewritten to run on a set of P^2 processors using message passing code of the kind described. We have simplified matters by assuming that the processors have been organized into a two-dimensional array *PROC*(P,P) and that the processor array elements may be addressed for the purpose of exchanging data items: nor-

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

C CODE FOR PROCESSOR (P1,P2)

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

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

C SEND DATA TO OTHER PROCESSORS

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

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

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

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

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

40    CONTINUE
        ...
```

FIGURE 2 Jacobi relaxation code parallelized manually.

mally, a structure of this kind would have to be set up by the user first, and references would have to be converted to those provided by the environment. Further, we assume that the array sizes are multiples of P. Optimization of communication has been performed insomuch as messages have been extracted from the loops and organized into vectors for sending and receiving. When communication and computation are overlapped, as could be done here by carefully arranging the order in which local data is updated, the resulting code is considerably longer.

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

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

3 THE VIENNA FORTRAN LANGUAGE

3.1 The Programming Model

Vienna Fortran assumes that a program will be executed by a machine with one or more processors according to the SPMD programming model as described above. This model requires that each participating processor execute the same program; parallelism is obtained by applying the computation to different parts of the data domain simultaneously. The generated code will store the local parts of arrays and the overlap areas locally and use message passing, optimized where possible, to exchange data. It will also map logical processor structures declared by the user to the physical processors which execute the program. These transformations are, however, transparent to the Vienna Fortran programmer.

3.2 The Language Features

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

1. The **processors** which execute the program may be explicitly specified and referred to. It is possible to impose one or more structures on them.
2. The **distributions** of arrays can be specified using annotations. These annotations may use processor structures introduced by the user.
   (1) Intrinsic functions are provided to specify the most common distributions.
   (2) Distributions may be defined indirectly via a map array.
   (3) Data may be replicated to all or a subset of processors.
   (4) The user may define new distribution functions.
3. An array may be **aligned** with another array, providing an implicit distribution. Alignment functions may also be defined by the user.
4. The distribution of arrays may be changed dynamically. However, a clear distinction is made between arrays which are statically distributed and those that may change at runtime.
5. In **procedures**, dummy array arguments may inherit the distribution of the actual argument or be explicitly distributed, possibly causing some data motion.
6. 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.
7. Arrays in **common blocks** may be distributed.
8. **Allocatable** arrays may be used in much

the same way as in Fortran 90. Array sections are permitted as actual arguments to procedures.

9. **Assertions** about relationships between objects of the program may be inserted into the program.

Vienna Fortran does not introduce a large number of new constructs, but those it does have are supplemented by a number of options and intrinsic functions, each of which serves a specific purpose. They enable the user to exert additional control over the manner in which data is mapped or moved, or the code is executed. An overview of the Vienna Fortran language extensions for FORTRAN 77 is given below.

We use terminology and concepts from the definition of FORTRAN 77 (and, occasionally, Fortran 90) freely throughout.

3.3 The Language Extensions: An Overview

Vienna Fortran includes all of the following language extensions to FORTRAN 77. Many of them will be discussed in the examples below, where their use is further described in an informal manner. For a complete and precise description of the language, see Zima et al. [11]. The reader is also referred to Chapman et al. [12] for further examples of the use of these extensions and demonstration of their expressiveness.

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 the column-major ordering of array elements defined 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:$NP)**, is always implicitly declared and may be referred to. This is the default primary array if there is no processor statement in a program. The index of an executing processor in **$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
```

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.

Distribution Intrinsics

Direct distributions may be specified by using the elision symbol, as described above, and the *BLOCK* and *CYCLIC* intrinsic functions. The *BLOCK* 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.

The linear expressions which specify an alignment may contain, in addition to the usual arithmetic operators "+", "−", and "*", the intrinsic functions *MAX*, *MIN*, *MOD*, *LBOUND*, *UBOUND*, and *SIZE*. The latter three are intrinsic functions similar to Fortran 90, and refer to the lower bound, upper bound, and size of an array (dimension), respectively.

The *INDIRECT* distribution intrinsic function enables the specification of a mapping array which allows each array element to be distributed individually to a single processor. The mapping array must be of the same size and shape as the array being distributed. The values of the given array are processor numbers (in **$P**):

```
INTEGER IAPROCS(1000)
REAL A(1000) DIST(INDIRECT(IAPROCS))
```

Thus, for example, the value of *IAPROCS(60)* is the number of the processor to which *A(60)* is to be mapped. Note that *IAPROCS* must be defined before it is used to specify the distribution of *A*, and that each element of *A* can be mapped to only one processor.

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))
```

The distribute statement begins with the keyword **DISTRIBUTE** and a list of the arrays which are to be distributed at runtime. Following the separator symbol "::", a direct, implicit, or indirect distribution is specified using the same constructs as those for specifying static distributions. It has an optional **NOTRANSFER** clause; if it appears, then it specifies that the arrays to which it applies are to be distributed according to the specification, but that old data (if there is any) is not to be transferred. Thus only the access function is modified. For example:

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

in the above statement, both arrays *A* and *B* are redistributed with the new distribution *CYCLIC(10)*, however for the array *B* only the access function is changed, the old values are not transferred to the new locations. Whenever an array is redistributed via a distribute statement, then any arrays connected to it are also automatically redistributed to maintain the relationship between their distributions.

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. The distribution expressions consist of one or more distribution functions (which may contain arguments such as a length), or a "*" which matches any distribution. The distribution of an array is matched only if it is matched in all dimensions. The first case which satisfies the actual distributions of the selected arrays is chosen and its statements 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 different arrays may be compared in a similar manner within an IF statement.

Allocatable Arrays

An array may be declared with the allocatable attribute by specifying the keyword **ALLOCATABLE** as in Fortran 90. The declaration defines the rank of the array, but not the bounds of any dimension. The array may be statically or dynamically distributed. 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 unless it is currently allocated and a distribution has been associated with it. The allocatable attribute should be used wherever the size of an array is not known at compile time; the user is thus able to distribute the array with its actual bounds, rather than distributing the largest array which is permitted. Further, it may remove the need for work arrays in some situations.

Common Blocks

Common blocks in which no data is explicitly distributed have the same semantics 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 dynamically distributed. Once storage space has been determined for a named common block, then it may not change during program execution. Note that, in accordance with Fortran 90, allocatable arrays may not be in common blocks.

Procedures

Dummy array arguments may be distributed in the same way as other arrays. If the given distribution 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 be 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.

Intrinsic Functions

A number of intrinsic functions from Fortran 90 are very useful for writing programs distributed memory machines. They include the functions *SIZE*, *LBOUND*, *UBOUND*, *COUNT*, *ANY*, and *ALL*, which may be used in Vienna Fortran programs.

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 (cf. *ADD* below); the result is not available until the end of the loop. The user may also define reduction functions for operations which are commutative and associative in the mathematical sense. The intrinsic reduction operators provided by Vienna Fortran are *ADD*, *MULT*, *MAX*, and *MIN*. The following statement results in the values of the array *A* being summed and the result being placed in the variable *X*.

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

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.

4 WRITING PROGRAMS IN VIENNA FORTRAN

In this section we introduce many of the language extensions of Vienna Fortran by showing how they may be used to produce parallel code for some simple problems. We discuss several different issues related to programming in general. The ideas in this section could, in principle, be applied to other programming languages which use similar data structures.

4.1 Distributing Data to Processors

In Section 2 above, we saw how a Jacobi relaxation might be parallelized manually, under certain simplifying assumptions. We present two versions of this same code in Vienna Fortran. The first version tends to run faster on machines with a high communication latency, whereas the second version will often be preferred for its overall communication behavior.

All that has been added to the sequential code to produce the first parallel Jacobi relaxation, shown in Figure 3, is an annotation which tells the compiler to distribute the second dimension of all three arrays by block to all processors: the compiler will generate code to place the data accord-

```
C PARALLEL CODE VERSION 1

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

      CALL INIT (U, F, N)

        ...

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

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

 40      CONTINUE

           ...
```

FIGURE 3 Jacobi relaxation code in Vienna Fortran.

ingly. It is also responsible for inserting the necessary communication.

Note that no reference has been made to the processors executing the program in this example. Thus the data is mapped implicitly to a one-dimensional processor array consisting of the processors available at runtime. The elision symbol was used to ensure that only one dimension of the arrays is distributed.

An alternative implementation of the Jacobi relaxation requires that the arrays be mapped to a two-dimensional processor grid. It begins with the following declarations:

```
C Jacobi relaxation code in Vienna Fortran:
                                      version 2
   ASSERT(NP .GE. 4)
        PROCESSORS P(NP, NP)
 REAL UNEW(1:N, 1:N), U(1:N, 1:N), F(1:N, 1:N)
 &                        DIST(BLOCK, BLOCK)
        . . .
```

The rest of the code is the same as shown in Figure 3. This Vienna Fortran program first declares a square processor array, whose size will be determined at load time. The programmer requires at least four processors in each dimension and expresses this by making an appropriate assertion. The array declaration includes an annotation to distribute the arrays by block in both dimensions: this maps them in square blocks to the processors. The code has been written so as to be independent of the number of processors it will execute on, and does not need to be recompiled each time it runs on a different configuration. (But, if it is to be run on a fixed number of processors every time, then a processor array may naturally be declared with fixed bounds—it is likely to result in faster code). This is the data distribution used in the manually parallelized version of the code and the compiler must distribute the data and organize the communication to produce the code similar to that shown in Figure 2.

This version of the code will thus result in compiled code which is markedly different from the first version and may exhibit different behavior at runtime. When the first version is executed, the data are distributed in blocks of columns to the processors. To compute local values of *UNEW*, a processor will require a vector of values from the two neighboring processors. The second version distributes data in squares. As a result, a processor will require values from four neighboring processors to compute its local values. In general, the second version requires fewer data items to be sent and received, however the number of messages per iteration increases from two to four. Thus, the actual performance of the codes will be dependent not only on the message transfer costs of the underlying hardware but also on the start-up time per message. It is an easy matter to implement both versions in Vienna Fortran and compare their performance.

Other Ways to Distribute Arrays

We have already seen the intrinsic functions provided by Vienna Fortran to specify the most common kinds of distributions: *BLOCK* and *CYCLIC* map a dimension of an array to a dimension of a processor array. The following are further examples of Vienna Fortran array declarations annotated by a distribution:

```
   PROCESSORS P2(NP, MP)
   REAL XX(1000, 100)
 &                   DIST(CYCLIC(50), BLOCK)
   REAL YY(10000) DIST(BLOCK) TO $P
   INTEGER KK(500, 50, 5)
 &         DIST(BLOCK, CYCLIC,:) TO P2/2, 1/
```

Arrays *XX* and *KK* are distributed to *P2*, however, the dimensions have been permuted in the second case, so that the first dimension of *KK* is distributed by block to the second dimension of *P2*, and the second dimension of *KK* is scatter distributed to the first dimension of *P2*. *YY* is distributed to **$P**, which has *NP*MP* elements in this case. Remember that the standard ordering of array elements defined in FORTRAN 77 may be applied to processor arrays, so that there is a well-defined relationship between the elements of **$P** and those of *P2*.

Implicit distribution, or alignment might be used, for example, to parallelize the following kernel.

The elements of arrays *X* and *Y* are aligned with the elements of array *ZX* in the example above: for each *I* from 1 through *N*, *X*(*I*) is mapped to the processor that owns element *ZX*(*I* + *10*). The *$* symbol is merely a placeholder, indicating that multiple arrays are being aligned. Note that the scalar variables are replicated.

```
      PARAMETER (N = . . .)
      REAL ZX(N + 12) DIST(BLOCK)
      REAL X(N), Y(N) ALIGN $(I) WITH
&                                ZX(I + 10)
      REAL Q, R, T
      DO 11 K = 1, N
          X(K) = Q + Y(K)* (R*ZX(K + 10)
&                          + T* ZX(K + 11))
11    CONTINUE
```

In practice, alignments can be used whenever there is a fixed relationship between two arrays that is of a very specific nature. In other situations it will generally suffice, or be more appropriate, to specify that data items are to be distributed "in the same way." In the above, for example, the distribution of *X* and *Y* could have been expressed by giving them the same distribution function as *ZX*:

```
        REAL X(N), Y(N) DIST(=ZX)
```

This distributes *X* and *Y* by block, with the appropriate block sizes. In this case, *X* and *Y* would be distributed evenly by block across the processors. Since they have fewer elements than *ZX*, the length of their blocks may be slightly smaller than the length of the blocks of *ZX*. When they are aligned with *ZX* as above, then the lengths of the first blocks of *X* and *Y* will be identical to those of *ZX*. However, the last processor will contain fewer elements of these arrays. For example, if $N = 100$ and the data is distributed to four processors, then the second distribution would distribute 25 elements of *X* and *Y* to each processor, whereas the alignment with *ZX* would result in the mapping of 28 elements of *X* and *Y* to the first three processors, and only 16 elements to the last of them. Thus the elements of *X* and *Y* are not spread evenly over the processors. It will depend very much on the nature of our computation which of these distributions performs better.

Note that if we choose to distribute *X* and *Y* in the same way as *ZX*, we could actually distribute them all by one single declaration in this case. But that would not be true in a subroutine when, say, *ZX* is a dummy argument whose distribution is not known. Both alignment and the referral to the distribution of other arrays are important in subroutines where information on the distribution of dummy arguments is incomplete.

Rather more complex distributions and alignments are required in many real applications. Many of them, such as arbitrary rectilinear block distributions, are useful to the programmer and can be efficiently implemented. We will see an example of a user-defined distribution function in Section 5.

4.2 Using Subroutines in Vienna Fortran

We discuss the main issues which arise when subroutines* are invoked with distributed arguments by, again, looking at a very simple example. This permits us to ignore the computational problem and concentrate on the situations a programmer will need to be able to deal with.

It is common practice to write subroutines for such operations as matrix multiplication, which are used frequently. In this section we consider how this is done in Vienna Fortran.

When a distribution annotation is appended to a declaration in Vienna Fortran, then that distribution has the same scope as the declaration itself. In a subroutine, both local arrays and dummy array arguments may be given an explicit distribution when they are declared. As we will see below, this makes the mechanism of appending distribution annotations to array declarations a very powerful tool, enabling a controlled redistribution of data.

One version of a subroutine to multiply matrices in Vienna Fortran is as follows:

```
      SUBROUTINE MATMUL(A, B, C, N, M, L)
      REAL A(N, M), B(M, L), C(N, L) DIST(*)
      DO 30 I = 1, N
         DO 30 J = 1, L
            C(I, J) = 0.0
            DO 30 K = 1, M
               C(I, J) = C(I, J) + A(I, K)*B(K, J)
30    CONTINUE
      RETURN
      END
```

In this routine we employ the additional method for specifying distributions which can be

* We will not examine functions separately; they can be written similarly.

used for dummy array arguments only. If a "*" is used to specify the distribution, then the dummy argument inherits the distribution of the actual array. This means that each time the above routine is called, the actual arguments may be distributed differently to the processors. Interprocedural distribution analysis will often reveal the distribution functions which reach the subroutine, and the compiler is then able to generate code based on that information. This is a flexible way to write subroutines. But an unfortunate consequence of using inherited distributions is that the compiler may not always have precise (or, if it is separately compiled, any) information on the actual distributions which may reach the dummy arguments. In cases where this analysis fails, there is a way of providing extra help. If the user knows that only a few distributions will occur, then this information may be provided in a **RANGE** clause which is appended to the distribution. For example, the specification:

```
 REAL A(N, M) DIST(*),
&        RANGE((BLOCK, BLOCK),
&                        (BLOCK, CYCLIC(100)))
```

declares that only the distributions (*BLOCK*, *BLOCK*) and (*BLOCK*, *CYCLIC*(100)) are allowed for the dummy argument *A*.

Further, the efficiency of the computation within the subroutine may depend very heavily on the actual distributions of the arguments, thus yielding good performance in some cases and very poor performance in others.

An alternative implementation might distribute the dummy array arguments explicitly. We may write, for example:

```
SUBROUTINE MATMUL(A, B, C, N, M, L)
REAL A(N, M), C(N, L) DIST(BLOCK,:) TO $P
REAL B(M, L)
DO 30 I = 1, N
  . . .
```

Now this subroutine also has three dummy argument arrays, two of which, *A* and *C*, are distributed by block in the first dimension to all available processors whereas the third, *B*, is replicated. The dummy arguments are explicitly distributed in order to eliminate communication during the computation of the result. However, the actual arguments may not have the same distribution as the dummy arguments with which they are associated. When their distributions differ, they must be redistributed on entry to the subroutine to match the specified distribution. In general, their original distribution must also be restored on exit from the subroutine. Thus the efficient implementation of the computation within the subroutine has a price: the redistribution of actual arguments may sometimes be very costly.

We have seen the apparent difficulty in resolving two legitimate demands of a general purpose subroutine: that it handle a variety of different arguments, which may be differently distributed, on the one hand, and that it handle them efficiently on the other hand. Redistribution may be costly, yet we may want to implement the routine in a way that is handled optimally on the target machine. Vienna Fortran provides a construct which may be used in this situation: the **DCASE** construct, which is modeled along the lines of the CASE construct in Fortran 90. It enables the selection of a block of statements according to the actual distribution of one or more arrays.

The third subroutine for matrix multiplication begins as follows:

```
    SUBROUTINE MMUL(A, B, C, N, M, L)
    REAL A(N, M), B(M, L), C(N, L) DIST(*)
    INTEGER LEN, LSUB
    SELECT DCASE (C, A):
      CASE(BLOCK, :), (BLOCK, :)
        IF (M*L .LE. MAXSIZE) THEN
            CALL MATMUL(A, B, C, N, M, L)
        ELSE LEN = L / $NP
            DO 45 J = 1, $NP
              CALL MATMUL1
                (A, B, C, N, M, L, LEN, J)
45          CONTINUE
        ENDIF
      CASE(BLOCK, BLOCK), (BLOCK,*)
        . . .
      CASE DEFAULT
        . . .
    END SELECT
```

In the above, the matrix operation is handled in a specific way depending on how the actual argument arrays are distributed. In this way, we can insert appropriate code or call further subroutines as required. The compiler has precise information on the distribution functions of the selected arrays for the block of statements within the cases. Only one of the case alternatives is executed; if none of the other specifications match, then the default (if present) is selected. Here, the cases are examined in the order in which they occur textually. The first distribution expression is compared with the actual distribution of *C*, and the second with that of *A*. If *C* is distributed by block in the first dimension and not at all in the second, and *A* likewise, then the first case is selected and its code executed. Otherwise, the distribution of *C* is then compared with the next case: if it is distributed by block in both dimensions, then if *A* is distributed by block in the first dimension, this case is selected. An "*" matches any distribution whatsoever.

```
      PARAMETER (NX = 100, NY = 100)

      REAL U(NX, NY), F(NX, NY), V(NX, NY) 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

      CALL YSWEEP(V,NX,NY)

      DO 30 J = 1, NY
         DO 30 I = 1, NX
            U(I, J) = V(I, J)
30    CONTINUE
      ...

      SUBROUTINE YSWEEP (V,NX,NY)
      REAL V(NX,NY) DIST( BLOCK, : )

C     Sweep over y-lines
      DO 20 I = 1, NX
         CALL TRIDIAG( V(I, :), NY)
20    CONTINUE
```

FIGURE 4 An ADI iteration: Version 1.

5 APPLICATIONS IN VIENNA FORTRAN

In this section we look at the structure of two frequent kinds of codes that are used to handle a variety of applications. The first of them shows how a particular numerical method might be expressed in Vienna Fortran; the second code shows how one could approach problems which cannot be efficiently distributed at compile time. We then briefly discuss some issues which arise with certain Fortran constructs and programming styles.

5.1 ADI Iteration

One well known and effective method for solving partial differential equations in two or more dimensions is known as alternating direction implicit (ADI).[13] 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 and the same operation is then performed, again independently, on each y-line of the array.

We present two versions of a step of the ADI algorithm here. The first version is shown in Figure 4. Here, the current solution, *U*, the right hand sides, *F*, and the temporary array, *V*, are all distributed by blocks of columns to the implicit one-dimensional array of processors, **$P**.

In this version, the sweep over the columns (representing x-lines) is performed by the first loop while the sweep over the rows (representing y-lines) is performed via a call to the routine *YSWEEP*. In each case, the subroutine sequential *TRIDIAG* (not shown here) is given a right hand side and overwrites it with the solution of a constant coefficient tridiagonal system.

The array *V* is redistributed when subroutine *YSWEEP* is invoked; thus it is distributed in blocks of columns when the first loop is executed, and is distributed in blocks of rows when the second loop is performed. This makes it possible to use a sequential tridiagonal solver in each of these since neither x-lines in the first loop nor the y-lines in the second loop cross processor boundaries. Note that the redistribution of *V* 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. The communication here is contained implicitly in the subroutine call and the tridiagonal solvers themselves do not require interprocessor communication.

Since the distribution of a statically distributed array has to be restored on return to the calling unit, the array *V* is redistributed at subroutine exit to be distributed by columns. Hence, the assignment of the values of *V* to *U* in the last loop does not cause any communication.

We had presented another version of this algorithm in our earlier paper [12]; we reproduce the

```
      PARAMETER (NX = 100, NY = 100)

      REAL U(NX, NY), 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 10 I = 1, NX
         CALL TRIDIAG( V(I, :), NY)
10    CONTINUE

      DO 30 J = 1, NY
        DO 30 I = 1, NX
           U(I, J) = V(I, J)
30    CONTINUE
```

FIGURE 5 An ADI iteration: Version 2.

code here in Figure 5. In this second version, we do not call a subroutine to enforce a redistribution of *V*. Instead, *V* is declared to have a dynamic distribution, and is initially distributed by block in the second dimension. The *range attribute* specifies that the only distributions allowed for *V* are blocks of rows or columns. Thus, the situation for the first loop remains the same, that is, the columns do not cross processor boundaries and hence the sequential tridiagonal solver can be employed. After the first loop we explicitly redistribute the array *V* to be blocked by rows via a **DISTRIBUTE** statement. Now, the second loop ranges over the rows of *V* again using the sequential tridiagonal solver. In this code, the final assignment of the array *V* to the array *U* will also induce communication similar to the "transpose" at the subroutine boundary above since *U* and *V* are distributed in different dimensions. Thus in the first case we performed the communication implicitly, by passing the array to a subroutine where the dummy argument has an explicit distribution, and in the second case we executed a statement to do the same work.

There are many ways in which the ADI algorithm may be formulated. For example, another formulation would declare array *V* with a static distribution and not redistribute it at all. A parallel tridiagonal solver would then be called in the second loop; the communication would take place within the solver. Similarly, one could declare a two-dimensional processor structure and distribute the arrays by block in both dimensions: a parallel tridiagonal solver would then be used for both the x- and the y-lines.

All versions of this algorithm are equally easy to express in Vienna Fortran: which of these performs the best may be dependent on various factors including message startup and transfer times of the underlying architectures. The point is that 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 and thus experiment with the different versions. In marked contrast, such changes will typically induce weeks of reprogramming in a message-passing language.

5.2 Irregular Distributions

There are a number of scientific codes where an efficient distribution of some of the major data structures is not possible at compile time. The distribution of an array may depend, for example, on the values of another array—or even on its own values, as in the example given below. 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.

Here, we look at an abstraction of a two-dimensional unstructured mesh Euler solver. The mesh is represented by triangles and the flow variables are stored at the vertices of the mesh. We reproduce only one part of the computation, which consists of accumulating at each node the contribution from each of the edges incident upon it. The computation is implemented as a loop over the edges: the contribution of each edge is subtracted from the value at one node and added to the value at the other node.

Figure 6 shows one way in which this computation may be specified in Vienna Fortran. The mesh is represented by the array *EDGE*, where *EDGE*(*I*, 1) and *EDGE*(*I*, 2) are the node numbers at the two ends of the *I*th edge. The arrays *X* and *Y* represent the values at each of the *NNODE* nodes.

Consider the distribution of the data across the (implicit) one-dimensional array of processors. Since the mesh must be distributed at runtime, in order to balance the computational load across the processors, each of the arrays has to be dynamically distributed.

The array *X*, representing a data value at each node, is declared to be dynamically distributed with an initial block distribution. Further below, this array is explicitly distributed via the indirect distribution mechanism provided by Vienna Fortran. The indirection is based on the mapping ar-

```
      PARAMETER (NNODE = ... )
      PARAMETER (NEDGE = ... )

      REAL X(NNODE) DYNAMIC, DIST( BLOCK)
      REAL Y(NNODE) DYNAMIC, CONNECT (=X)
      INTEGER MAP(NNODE) DIST( BLOCK)
      REAL EDGE(NEDGE,2) DYNAMIC, DIST( BLOCK)
       ...
      CALL PARTITION( MAP, EDGE )

      DISTRIBUTE X :: ( INDIRECT(MAP)) NOTRANSFER (Y)
      DISTRIBUTE EDGE :: ( FDIST(MAP, EDGE, NEDGE, NNODE)
       ...
      FORALL I = 1, NEDGE  ON  OWNER( EDGE(I,1) )
        INTEGER N1, N2
        REAL  DELTAX

        N1 = EDGE(I,1)
        N2 = EDGE(I,2)

        DELTAX = F(X(N1), X(N2))

        REDUCE( ADD , Y(N1), - DELTAX)
        REDUCE( ADD , Y(N2), DELTAX)
      END FORALL
       ...
      END

      ...

      DFUNCTION  FDIST(MAP, EDGE, N, M)
      TARGET  A(N,*)
      REAL MAP(M) DIST(*)
      INTEGER EDGE(N,2) DIST(*)

      DO 10 I = 1, N
        A(I,:) DIST  TO  $P (MAP(EDGE(I,1)) )
   10 CONTINUE
      END
```

FIGURE 6 Code for unstructured mesh.

ray *MAP*, whose values are dependent on the structure of the mesh and are defined in the user specified routine *PARTITION* (the code for *PARTITION* has not been shown here). The value of the *I*th element of the array *MAP*, which must be declared with the same size as *X*, is the number of the processor in **$P** to which the *I*th element of the array *X* is distributed.

Y is also declared with the keyword **DYNAMIC** and is assigned the same distribution as *X*; its distribution is, however, connected with that of *X* by the **CONNECT** attribute. This means that when *X* is redistributed, *Y* is automatically redistributed with exactly the same distribution function. The **DISTRIBUTE** statement for array *X* specifies the **NOTRANSFER** attribute for array *Y*. This means that when the two arrays are redistributed, only the values of *X* are to be transferred to the new locations; the old values of *Y* are not moved.

The array *EDGE* is also declared with a dynamic distribution and is initially distributed by block. Given the structure of the computation, it would be useful to distribute *EDGE* in such a way that the values at one or both of its nodes are on the same processor. We have chosen to distribute the elements of *EDGE* to the processor which owns the values for the first of its nodes. Such a distribution cannot be described by the intrinsic functions, so it is specified by the user-defined distribution function (**DFUNCTION**) *FDIST* in Figure 6.

DFUNCTIONs are similar to regular Fortran functions, but have a special implicit argument declared with the keyword **TARGET**. It represents the array that is being distributed. Here, the distribution function *FDIST* takes as arguments the arrays *MAP* and *EDGE* and the special argument *A*. The function body then specifies that the *I*th row of the array *A* is to be distributed to the processor whose number is given by *MAP*(*EDGE*(*I*, 1)). Thus, when the distribution function *FDIST* is accessed in the distribute statement, the special argument *A* is associated with the array being distributed, i.e., *EDGE*, so that *EDGE* is distributed as required.

The computation is specified using a **FORALL** loop, with an **ON** clause to specify where each iteration is to be performed. Thus the iterations of the loop, over the edges in this case, can be executed in parallel. In Figure 6, the **ON** clause specifies that the *I*th iteration should be performed on the processor that owns the (*I*, 1)th element of *EDGE*. Nonlocal values which are read can be gathered before the execution commences.

The variables *N*1, *N*2, and *DELTAX* declared within the **FORALL** loop are private variables. Thus assignments to these variables do not cause flow dependencies between iterations of the loop. For each edge, the *X* values at the two incident nodes are read and used to compute the contribution *DELTAX* for the edge. This contribution is then accumulated into the values of *Y* for the two nodes.

Since multiple iterations will accumulate *Y* values at each node, different iterations write to the same array elements, which is not permitted within a **FORALL**. So that this situation does not prevent parallel execution, Vienna Fortran provides special reduction statements which allow accumulations across the iterations of a **FORALL** loop. The reduction operator *ADD* is used here to accumulate the contribution of the edge to the values at the nodes on which it is incident. The results cannot be accessed within the **FORALL** loop, and hence the accumulations can be easily performed by the system after all iterations are completed. This code makes use of the reduction operator *ADD*.

The most important feature of this code as far as its compilation is concerned is that the values of *X* and *Y* are accessed via the edges, hence a level of indirection is involved. We distributed the arrays in such a way that the values at the first node of an edge are always local to a loop iteration, but the values at the second node may not be. The data distribution of each of the arrays is determined at runtime; thus the compiler cannot detect which references are local and which are not. In such situations, runtime techniques such as those developed in other projects [6, 14] are needed to generate and exploit the communication pattern.

5.3 Some Fortran Issues

There are several important features of Fortran codes which have not been dealt with in the sections above. We discuss just a few of them.

Common Blocks

Common blocks are used in FORTRAN 77 to enable different program units to define and reference the same data without using arguments, and to share storage units. In Vienna Fortran, the user may retain full FORTRAN 77 semantics for a common block by not explicitly distributing any of the objects within it at any place in the program. In this case, there is conceptually one copy of the common block, and conventional storage association holds for it. Note that, in accordance with the rules of Fortran 90, allocatable arrays may not be in common blocks. Vienna Fortran also permits explicit distribution of arrays within named common blocks. However, their distribution may not be dynamic. If distributions are given at more than one place in the program for objects in common blocks with the same name, then they must be identical except for the names of the objects. The common block storage sequence holds for those parts of a common block which are not explicitly distributed—we refer to these as replicated sections below. For example:

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

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

Some restrictions should be placed on the use of the Fortran EQUIVALENCE statement when data objects are distributed. In Vienna Fortran, we do not permit an implicit distribution by equivalencing. Further, no distributed array may be associated by equivalence with any other distributed object. Thus equivalence association is permitted between replicated data only.

Work Arrays

FORTRAN 77 does not permit dynamic storage allocation. It is thus common programming practice to declare arrays with a maximum size and use them with some other, smaller, size during the computation. Further, a large work array is often declared, parts of which are then used as individual arrays with the size and shape required by the computation. So that arrays may be declared as they are used, Vienna Fortran includes the concept of allocatable arrays as defined in Fortran 90. An individual array with unknown size may be declared with the **ALLOCATABLE** attribute. Once its bounds are known, it can be allocated using the **ALLOCATE** statement. An allocatable array may also be annotated with distribution expressions to specify the distribution of the array. This distribution expression can be completely evaluated only after the allocation of the array. For example:

```
REAL A(:) ALLOCATABLE DIST(BLOCK)
    . . .
READ (*,*) LEN
ALLOCATE (A(LEN))
```

Here we have declared *A* to be a one-dimensional allocatable array. Thus the distribution expression (*BLOCK* in this case) will be evaluated for *A* with length *LEN*, and the *LEN* elements of *A* distributed evenly across the processors. Without allocatable arrays, *A* would have to be declared with some maximum size (greater than *LEN*) and distributed by *BLOCK* with respect to this maximum size. Since only the first *LEN* elements of *A* are to be used, some of the processors might not have any of the elements of *A* which are actually used in the computation. By using allocatable arrays, we make sure that all processors are involved in the computation.

As noted above, many Fortran applications are characterized by the fact that runtime data determines the size of the underlying data objects. In many applications, the actual number of objects involved is also unknown at compile time or may vary during computation. Such situations require the work array to be distributed dynamically, since the actual distribution of the objects may be dependent on runtime data. Such an array is declared with the **ALLOCATABLE** and **DYNAMIC** attributes. One strategy for distributing such a work array is to distribute each of these objects independently to all processors, by *BLOCK* for example. Another strategy would be to distribute each of these objects to a subset of processors. This kind of distribution must be handled by a user-defined distribution function in Vienna Fortran.

6 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) [15]. Languages for coordinating individual threads of a parallel program, such as LINDA [16] and STRAND [17], 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 [18] in the context of shared memory systems with nonuniform access times. This research was continued in the Kali programming language [19] 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 [20, 21], 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 nonlocal accesses. In Booster [22, 23], 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 that are currently being developed specify the distribution of arrays in three stages: alignment, partition, and a physical map [1]. 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 [4], 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 runtime. Fortran 90D [24], proposed by researchers at Syracuse University, is based upon CM Fortran [25].

Digital Equipment Corporation has proposed language extensions[2] 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) [3] 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 [26]. 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) [8] 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 [6] 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 [9] and ASPAR [27] (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 [28] 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 [29], Crystal [30]) have also been developed which are targeted to distributed memory machines.

Others [31–33] compile languages based on SIMD semantics. These attempt to minimize the interprocessor synchronizations inherent in SIMD execution. The AL compiler [34], 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 the intercell communications.

The PARTI primitives, a set of runtime library routines have been developed to handle irregular computations [14, 35]. These primitives have been integrated into the Vienna Fortran Compilation System 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 [6, 36]. Explicit runtime generation of messages is also performed by other researchers [29, 30, 32]; however, these do not save the extracted communication pattern to avoid recalculation.

7 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 [8]. 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:

1. Static array distributions
2. Arbitrary rectilinear block distributions
3. Inherited distributions for dummy array arguments
4. Forall loops

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

1. Interprocedural communication analysis
2. Communication optimization: matching access patterns to aggregate communication routines, elimination of redundant communication, and fusion of communication statements
3. Interprocedural dynamic distribution analysis
4. Interprocedural distribution propagation
5. Procedure cloning
6. Optimization of parallel loop scheduling
7. Optimization of irregular access patterns, based on the PARTI routines [15]

The current 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.

8 CONCLUSIONS

In view of the increasing importance of distributed memory parallel computing systems, it is vital that the task of writing new programs and converting existing (sequential) code to these machines be greatly simplified. An approach which may substantially reduce the cost of developing codes is to provide a set of language extensions for existing sequential languages (in particular, Fortran and C) that are not bound to any specific existing system but can be used across a wide range of architectures. These extensions should be as simple as possible, but they should also be broad enough to permit the expression of a wide variety of algorithms at a high level. In particular, since the data distribution has a critical impact on the performance of the program at runtime, tight programmer control of the mapping of data to the system's processors must be possible.

We believe that Vienna Fortran is a significant step on the path towards a standard in this area.

The authors would like to thank Peter Brezany, Andreas Schwald, Mark Furtney, Irene Qualters, Joel Saltz, John Van Rosendale, and the Fortran D group at Rice University for their helpful comments and discussions.

REFERENCES

[1] M. Chen and J. Li, "Optimizing Fortran 90 Programs for Data Motion on Massively Parallel Sys-

tems." *Technical Report YALE/DCS/TR-882*, Yale University, Jan. 1992.

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

[3] D. Pase, "MPP Fortran Programming Model," *High Performance Fortran Forum*, Houston, TX, January 1992.

[4] 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.

[5] G. Steele, "Proposals for Amending High Performance Fortran," *High Performance Fortran Forum*, Houston, TX, January 1992.

[6] C. Koelbel and P. Mehrotra, "Compiling global name-space parallel loops for distributed execution, *IEEE Trans. Parallel and Distributed Syst.*, vol. 2(4), pp. 440–451, 1991.

[7] H. M. Gerndt, "Automatic parallelization for distributed-memory multiprocessing systems," PhD thesis, University of Bonn, December 1989.

[8] H. Zima, H. Bast, and M. Gerndt, "Superb: A Tool for Semi-Automatic MIMD/SIMD Parallelization," *Parallel Comput.*, vol. 6, pp. 1–18, 1988.

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

[10] S. Benkner, B. Chapman, and H. Zima, "Vienna Fortran 90," *Proceedings of the SHPCC Conference, 1992*, to appear.

[11] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald, "Vienna Fortran—a Language Specification," *ICASE Internal Report 21, ICASE*, Hampton, VA, 1992.

[12] B. Chapman, P. Mehrotra, and H. Zima, *Languages, Compilers, and Run-time Environments for Distributed Memory Machines*. New York: Elsevier, pp. 39–62.

[13] G. I. Marchuk, *Methods of Numerical Mathematics*. New York: Springer-Verlag, 1975.

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

[15] D. Pountain, *A Tutorial Introduction to Occam Programming*. Colorado Springs, CO: Inmos, 1986.

[16] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and friends," *IEEE Computer*, vol. 19, pp. 26–34, 1986.

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

[18] C. Koelbel, P. Mehrotra, and J. Van Rosendale, "Semi-automatic process partitioning for parallel computation, *Int. J. Parallel Programming*, vol. 16, pp. 365–382, 1987.

[19] P. Mehrotra and J. Van Rosendale, *Advances in Languages and Compilers for Parallel Processing*. Cambridge, MA: Pitman/MIT Press, 1991, pp. 364–384.

[20] M. Rosing, R. W. Schnabel, and R. P. Weaver, "Expressing Complex Parallel Algorithms in DINO," *Proceedings of the 4th Conference on Hypercubes, Concurrent Computers, and Applications*. Los Altos, CA: Golden Gate Enterprises, 1989, pp. 553–560.

[21] 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.

[22] E. Paalvast and H. Sips, "A high-level language for the description of parallel algorithms," *Proceedings of Parallel Computing 89*, Leyden, The Netherlands, North-Holland, August 1989, pp. 467–472.

[23] E. Paalvast, A. van Gemund, and H. Sips, "A Method of Parallel Program Generation With an Application to Booster Language," *Proceedings of the 4th International Conference on Supercomputing*, Amsterdam, June 1990. New York: ACM Press, 1990, pp. 457–469.

[24] M. Wu and G. Fox, "Fortran 90D Compiler for Distributed Memory MIMD Parallel Computers," *Technical Report SCCS-88b*, Syracuse University, New York, 1991.

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

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

[27] K. Ikudome, G. Fox, A. Kolawa, and J. Flower, "An Automatic and Symbolic Parallelization System for Distributed Memory Parallel Computers," *Proceedings of the Fifth Distributed Memory Computing Conference*, Charleston, SC, April 1990. Los Alamitos, CA: IEEE Computer Society Press, 1990, pp. 1105–1114.

[28] F. André, J.-L. Pazat, and H. Thomas, "PANDORE: A System to Manage Data Distribution," *International Conference on Supercomputing*, June 1990, pp. 380–388.

[29] A. Rogers and K. Pingali, "Process Decomposition Through Locality of Reference," *Conference on Programming Language Design and Implementation*, ACM SIGPLAN, June 1989. New York: ACM Press, 1989, pp. 69–80.

[30] J. Li and M. Chen, "Generating Explicit Communication From Shared-Memory Program References," *Proceedings of Supercomputing '90*, New York, NY, November 1990. New York: ACM Press, 1990, pp. 865–876.

[31] P. Hatcher, A. Lapadula, R. Jones, M. Quinn,

and J. Anderson, "A Production Quality C* Compiler for Hypercube Machines," *3rd ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, April 1991. New York: ACM Press, 1991, pp. 73–82.

[32] 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.

[33] A. P. Reeves, "Paragon: A Programming Paradigm for Multicomputer Systems," *Technical Report EE-CEG-89-3*, Cornell University, Ithaca, NY, January 1989.

[34] P. S. Tseng, "A Systolic Array Programming Language," *Proceedings of the Fifth Distributed Memory Computing Conference*, April 1990. Los Alamitos, CA: IEEE Computer Society Press, 1990, pp. 1125–1130.

[35] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman, "Run-time scheduling and execution of loops on message passing machines," *J. Parallel and Distributed Comput.*, vol. 8(2), pp. 303–312, 1990.

[36] C. Koelbel, P. Mehrotra, and J. Van Rosendale, "Supporting Shared Data Structures on Distributed Memory Architectures, *2nd ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, March 1990. New York: ACM Press, 1990, pp. 177–186.

[37] C.-A. Thole, "PACT—Parallel Architecture Compiler Target," GENESIS Working Paper B4.T14, SUPRENUM GmbH, Bonn, Germany, September 1990.