VIENNA FORTRAN – A FORTRAN LANGUAGE EXTENSION FOR DISTRIBUTED MEMORY MULTIPROCESSORS*

Barbara M. Chapman[a], Piyush Mehrotra[b] and Hans P. Zima[a]

[a]Department of Statistics and Computer Science, University of Vienna, Rathausstrasse 19/II/3, A1010 Vienna AUSTRIA

[b]ICASE, MS 132C, NASA Langley Research Center, Hampton VA. 23666 USA

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. However, 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 placement of data. In this paper, we present the basic features of Vienna Fortran along with a set of examples illustrating the use of these features.

1 Introduction

In recent years, a number of distributed memory multiprocessing computers have been introduced into the market (e.g. Intel's iPSC series, NCUBE, and several transputer based systems). In contrast to shared memory machines, these architectures are less expensive to build and are potentially scalable to a large number of processors. However, the associated programming paradigm requires the user to specify complete details of the synchronization and the communication of data between processors as dictated by

*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 23666. The authors assume all responsibility for the contents of the paper.

the algorithm. Experience has shown that forcing the user to provide such low level details not only makes distributed memory programming tedious and error prone but also inhibits experimentation.

The apparent advantages of using a shared memory programming paradigm have led to a number of attempts at simulating shared memory on such machines. These efforts range from providing a suitable combination of hardware mechanisms and operating system support (e.g. automatic paging and conflict resolution), to automatic parallelization.

Research in the last of these areas has resulted in the development of a number of prototype systems, such as Kali [10], SUPERB [6, 26], and the MIMDizer [16]. These systems allow the code to be written using global data references, as one would do on a shared memory machine, but require the user to specify the distribution of the program's data. This data distribution is then used to guide the process of restructuring the code into an SPMD (Single Program Multiple Data) program for execution on the target 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.

In this paper, we present a machine-independent language extension for FORTRAN 77, called Vienna Fortran, which allows the user to write programs for distributed memory machines using global data references only. Since the distribution of data to the processors is critical for performance, the extensions described here permit the user to explicitly control the mapping of data onto the processors. Vienna Fortran supports a wide range of facilities for specifying data distributions, including distribution by alignment, where one array is mapped so that it has a fixed relationship with another array. It also supports dynamic redistribution of data. Frequently occurring distributions can be specified in a simple manner, whereas the full language permits complex mappings. The overall aim of the language extensions provided by Vienna Fortran is to make the transition from the sequential algorithm to a parallel version as easy as possible, without sacrificing performance.

In this paper, we concentrate on describing the syntax and semantics of the basic language. Future papers will give a full description of both the language features and the compilation model. The next section introduces the Vienna Fortran extensions and supplements their description by several short examples. The use of the language constructs is more fully illustrated in Section 3, where three examples are presented and discussed. This is followed by a discussion of related work. Finally, in the last section we reach some conclusions.

2 The Basic Features of Vienna Fortran

The critical issue in programming distributed memory machines is the distribution of data across the processors of the target machine. An appropriate data distribution is crucial for the performance of the resulting parallel code; factors influencing the choice include the size of an application, the data access patterns, the number of processors available, details of the hardware and the characteristics of the communication mechanisms provided. Vienna Fortran provides an extensive set of language constructs which allow the user complete control over the mapping of the data structures in the program. These language extensions can be divided into a basic set of features which allow the user to handle the most frequently occurring but simple distributions and an extended set which provides more general mechanisms for mapping data structures onto arbitrary subsets of processors. In this section, we describe only the basic set of language constructs; the full language, together with a formal programming model, is specified in [27].

2.1 Processor Structure

The Vienna Fortran programming model requires a set of processors onto which the data can be distributed. Processor arrays can be declared in the program in a manner similar to Fortran arrays as shown here:†

PROCESSORS P2D(R, R) **ASSERT** R **.GE.** 8

The above statement declares a two dimensional processor array, $P2D$, with R^2 processors where the value of R is asserted to be greater than or equal to 8. Here, R is considered to be a constant whose value is determined at load time depending on the total number of processors available in the underlying machine. This allows the code to be parameterized by the number of processors and thus avoids recompilation if the number of processors change. The processor bounds can be compile-time constants if the code has to run on a fixed number of processors.

The R^2 processors introduced in the above declaration can be accessed in the same way as a two-dimensional array by the use of subscripts; for example P2D(3,7) determines a well-defined individual processor. Note, however, that the declaration does not imply a specific topology: in particular, it does **not** specify that the target processors are connected by a two-dimensional mesh.

The **primary processor structure**, as declared above, can be reshaped to lower dimensional **secondary processor structures** as follows:

PROCESSORS P2D(R, R) **ASSERT** R **.GE.** 8 **RESHAPE** P1D(R*R)

Here, $P1D$ is a one-dimensional processor array which provides an alternative view of the two-dimensional array $P2D$. The Fortran column-major ordering is used to establish a relationship between the primary and secondary processor structures.

The primary process structure of a program is implicitly associated with the built-in identifier **$P** while a reference to the parameterless intrinsic function **$NP** yields the total number of processors being used by the program during the current execution. An implicit one-dimensional structure declaration **$PL**(1:**$NP**) is also provided with each program. If there is no explicit processor declaration, then **$P** and **$PL** both are implicitly declared as one-dimensional identical arrays with the subscript range (1:**$NP**).

†In this paper, keywords in code segments are emboldened while comments are in italics.

Thus, if the program only requires a one-dimensional array of the maximum number of processors currently available on the target machine, then the processor declaration can be omitted.

2.2 Distribution of Arrays

An array declaration can be annotated to specify the distribution of its elements onto the program's processors. Distributed arrays in Vienna Fortran can be divided into two categories, *static* and *dynamic*. The former category consists of arrays whose distribution is fixed within the scope of the declaration. Arrays whose distribution may be modified during the execution of the program have to be declared **DYNAMIC** as discussed in the next subsection. This strict separation between arrays whose distribution remains static and others whose distribution maybe changed during program execution, facilitates the compiler's task of generating highly efficient code for the target machine.

The semantics of arrays for which no distribution has been specified is the same as that of distributed arrays, i.e., there is conceptually a single copy of the data structure. The compiler may optimize the implementation by replicating the data structure on each of the processors executing the program. It is the compiler's task to maintain consistency among these copies. Note that scalar variables are handled in a similar manner.

Static Distribution of Arrays

The distribution of arrays is specified by associating a *distribution expression* with the declaration of the array. For example,

REAL $ad_1, ad_2, \ldots, ad_r$ **DIST** dex

declares arrays, $ad_i, 1 \leq i \leq r$ and their associated subscript ranges while the distribution expression, dex, specifies how the arrays are distributed. All the arrays specified in one distribution declaration must have the same rank. In the basic language, the distribution expression for a n-dimensional array consists of n distribution functions each of which specifies how the corresponding dimension of the array is to be partitioned into disjoint sections. Intrinsic functions are provided for specifying the commonly occurring distributions such as block, cyclic and block-cyclic. The following declarations show the use of the intrinsic distribution functions for distributing data across a one-dimensional array of processors.

REAL A(100) **DIST** (**BLOCK**)
REAL B(100) **DIST** (**CYCLIC**)
REAL C(100) **DIST** (**CYCLIC** (K))
REAL D(100, 100) **DIST** (**BLOCK**, :)

Here A is partitioned by blocks such that each processor owns a contiguous block of elements, while the elements of the array B are distributed cyclically, i.e., in a round robin fashion, across the processors. The elements of array C are first partitioned into blocks of size K and then the blocks are cyclically assigned to the processors. The declaration for array D shows the use of the elision function, denoted by **:**, which "hides" a dimension of the array, i.e. specifies that the corresponding dimension is not to be distributed. Thus, the rows of the array D are partitioned into blocks with a block of rows being assigned to a processor - the columns are left undistributed.

The number of distributed dimensions of a distribution expression is called its *context rank*. In the basic language, the context rank of a distribution expression must match the rank of either the primary processor structure or that of one of the secondary processor structures as shown in the following examples.

```
PROCESSORS P2D(R, R)  ASSERT R  .LE. 8

REAL  A(100, 100)  DIST ( BLOCK, BLOCK )
REAL  B(100)  DIST ( CYCLIC )
REAL  C(100, 100)  DIST ( BLOCK, : )
REAL  D(100, 100, 100)  DIST (:, BLOCK, : )
```

Here, both the rows and the columns of the array A are partitioned by block across the two-dimensional processor array $P2D$, i.e., each processor is assigned a sub-block of the array. The array B is cyclically distributed across the processors of the one-dimensional processor array, **\$PL**, implicitly declared for each program. Similarly, the first dimension of the two-dimensional array C and the second dimension of the three-dimensional array D are distributed across the one dimensional array of processors.

The extended language provides more general facilities for distributing data arrays. In all the examples provided above, for instance, the data structures are mapped to the full processor array. The extended language also allows mappings to subsets of processors. Moreover, it removes the restriction on the context rank: the number of array dimensions which are distributed may be smaller than the number of dimensions of the processor array, permitting replication of a data array dimension across a processor array dimension.

Aligning Array Distributions: The Static Case

It is possible to declare the distribution of an array in terms of the distribution of another array, allowing the two arrays to be aligned to each other. The set of basic intrinsic *alignment functions* consists of **DISTRIBUTION**, **PERMUTE**, and **TRANSPOSE**. Intuitively, these functions can be thought of as yielding a distribution when applied to their argument(s).

The expression, **DISTRIBUTION**(A) yields the complete distribution of the array A. This can be syntactically replaced by "$= A$". The expression, **DISTRIBUTION**(A,d), yields the distribution of the dth dimension of the array A. This can be syntactically replaced by "$= (A.d)$".

The distributions of individual dimensions of an array can be permuted by the expression: **PERMUTE**(A, $perm$), where A is an array of rank n and $perm$ is a list $(/i_1, \ldots, i_n/)$ of length n, specifying a permutation of the numbers in the interval $1 : n$. Then the function reference results in the distribution which is obtained when the specified permutation is applied to the distribution expression of A, i.e., $(= (A.i_1), \ldots, = (A.i_n))$. The function, **TRANSPOSE**(A), is a special form of the permuting function applicable only to two dimensional arrays and yields the same effect as **PERMUTE**(A, $(/2, 1/)$).

The above alignment functions can be thus be used for distributing arrays as follows:

```
REAL A(100,100) DIST ( BLOCK, CYCLIC )
REAL B(100,100) DIST ( DISTRIBUTION(A) )
REAL C(100,100) DIST ( =A )
REAL D(100) DIST ( DISTRIBUTION(A,1) )
REAL E(100) DIST ( =(A.2) )
REAL F(100,100) DIST ( TRANSPOSE(A) )
```

Here, the arrays B and C are identically aligned with the array A. The distribution of the array D is the same as that of the first dimension of the array A, i.e., **BLOCK**, while E is distributed cyclically based on the distribution of the second dimension of A. The use of the transpose function declares the distribution of F to be (**CYCLIC**, **BLOCK**).

Note, that the extent of the array being aligned need not be the same as that of the original array as shown below:

```
REAL A(50) DIST ( BLOCK )

REAL B(11:60) DIST ( =A )
REAL C(80) DIST ( =A )
```

The array B is distributed across the processors "in the same way as A"; $B(11)$ will, for example, be placed on the same processor as $A(1)$, and the sizes of blocks of A and B will be the same. In contrast, when C is distributed "in the same way as A", then $A(1)$ and $C(1)$ will be placed on the same processor, but the lengths of the blocks will differ.

Dynamic Arrays

An array whose distribution may be modified during runtime must to be declared as dynamic. Such an array can then be used as the object of a distribute statement, (see subsection below), whenever its distribution needs to be changed. A dynamic array may have an optional initial distribution; if one is not given, the array may not be accessed (read or written) before it is distributed via a distribute statement.

```
REAL A(100, 100) DYNAMIC
REAL B(100) DYNAMIC DIST ( BLOCK )
```

The arrays A and B are declared as dynamic arrays with B having an initial block distribution.

Dynamic arrays may also have an optional *range attribute* which specifies the set of all distributions which can be associated with the arrays during the execution of the program. It consists of the keyword **RANGE**, followed by a parenthesized list of distribution expressions. If a distribute statement associates an array with a distribution not contained in the distribution range attribute, its effect is undefined. For example, in the declaration below,

REAL A(100,100) **DYNAMIC**
& **RANGE**((**BLOCK**, **BLOCK**), (**CYCLIC**, **CYCLIC**))

the dimensions of A can both be partitioned by either block or cyclic distributions only.

Aligning Dynamic Arrays

The alignment functions described earlier can be used to align the initial distribution of a dynamic array with another array. For example, the declaration

REAL A(100) **DYNAMIC** **DIST** (=B)

declares the initial distribution of the array A to be the same as that of B. However, this relationship is not maintained if either of the arrays is redistributed during the execution of the program. Note that in the above example, B can be either static or dynamic.

In contrast to the above situation, two dynamic arrays can be aligned to each other so that the alignment relationship remains valid throughout the execution of the program. This is done by providing a connect attribute in the declaration: it consists of the keyword **CONNECT** followed by a distribution expression based on the alignment functions as described earlier. For example, consider the following declarations:

REAL A(100) **DYNAMIC** **DIST** (**BLOCK**)
REAL B(100), C(100) **DYNAMIC** **CONNECT** (=A)

Here, the arrays B and C are declared to be *connected* to the array A via the identity alignment function. The array A is called the primary array, while B and C are secondary arrays. The connect set of A can be informally described as consisting of the array A together with all the secondary dynamic arrays aligned to it through connect attributes. The alignment relationships between the primary array and the secondary arrays in its connect set are maintained throughout the execution of the programs. A dynamic array can be a member of only one connect set while each connect set has only one primary array. Note that secondary arrays cannot be objects of a distribute statement, i.e., only primary arrays can be redistributed.

Distribute Statement

In contrast to static arrays, dynamic arrays may be the target of one or more distribute statements within the scope of their declaration. A distribute statement may appear at any place where an executable statement is permitted. The most general version of the distribute statement is as follows:

DISTRIBUTE $dg_1, \ldots, dg_s$

Here, $dg_1, \ldots, dg_s$ is a list of one or more *distribution groups*. Each *distribution group* has the form:

$$A_1, \ldots, A_r :: \; dex \; [notransfer\ attribute]$$

where the $A_i, 1 \leq i \leq r$, are arrays and *dex* is a distribution expression. In addition to the distribution expressions discussed up to now, *dex* may also use the **INDIRECT** intrinsic distribution function (see below).

Given the statement

DISTRIBUTE A :: *dex* [*notransfer attribute*]

the new distribution of A is determined by evaluating the distribution expression, *dex*, in the context of A. If A is a primary array of a connect set, then the distribute statement results in new distributions for each array in the connect set of A. The new distribution for each of the secondary arrays is determined by the new distribution of the primary array and the alignment relationship.

The *notransfer attribute* (which is optional) takes the form

NOTRANSFER $[(B_1, \ldots, B)]$

where $B_i, 1 \leq i \leq m$ are elements of connect set of A. If this list is omitted, the attribute applies to all elements of the connect set of A. The effect of this attribute is as follows: the values associated with the arrays B_i before execution of the distribute statement are ignored during execution of the statement, i.e., only the access function associated with the array is updated and no physical transfer of data takes place.

```
REAL  A(100, 100)  DYNAMIC

REAL  B(100, 100)  DYNAMIC  CONNECT(=A)
REAL  C(100, 100)  DYNAMIC  CONNECT( TRANSPOSE(=A))
        ⋮
DISTRIBUTE A :: ( BLOCK, CYCLIC(M) )  NOTRANSFER(C)
```

Here, A is the primary array while B is a secondary array which is identically aligned with A. The array C is also a secondary array but is aligned with A such that the distribution of the first dimension of A is the distribution of the second dimension of C and vice versa. When the distribute statement is executed, the current value of the variable M determines the size of the blocks of columns which are cyclically distributed across the processors. The same distribution is used for the array B, while a transposed distribution becomes the new distribution of the array C. The notransfer attribute specifies that the old data values of array C are to be ignored, whereas the old values of A and B are moved to their new positions.

INDIRECT Distribution

A special intrinsic distribution function called **INDIRECT** can be used for distributing dynamic arrays. This function allows each element of an array to be individually mapped to a processor. This is done via a **mapping array** as shown below:

```
INTEGER B(M)  DIST ( BLOCK )
REAL    A(M)  DYNAMIC
   ⋮
DISTRIBUTE  A :: INDIRECT(B)
```

Here B, the mapping array, is used to specify the distribution A. That is, the value of $B(i)$ specifies the number of the processor which owns element $A(i)$. Note that B must be completely defined before the distribute statement is executed.

2.3 Subroutines

Distribution annotations can be associated with formal parameter declarations in subroutines to specify how the arrays will be viewed and accessed within the subroutine. In addition, local arrays may either be distributed explicitly or aligned with a formal parameter. While Vienna Fortran usually accesses formal array parameters by the standard Fortran call-by-reference paradigm, there are situations (e.g. when an array slice 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.

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

If the formal parameter is declared dynamic, then no redistribution is required at subroutine entry unless an initial distribution is provided with the declaration. A dynamic formal parameter may be a target of an explicit **DISTRIBUTE** statement within the subroutine. The original distribution is restored if the actual argument is static or if the *restore attribute* has been specified.

For both static and dynamic formal parameters, a **NOTRANSFER** attribute can be specified with the declaration of the parameter. In such a situation, if a redistribution is required at subroutine 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 formal parameter are first going to be defined in the subroutine before being used. It is also appropriate if a dynamic array has not been distributed before the subroutine call.

A *nocopy attribute* (using the keyword **NOCOPY**) can be specified in the declaration of a formal parameter to suppress the generation of a subroutine-local copy of the formal array. In this case, the parameter transmission is by reference, i.e., the actual argument is directly accessed within the subroutine. For static formal parameters, it is an error if the distribution of the actual argument does not match the specified distribution. For dynamic formal parameters the actual argument may not be static. Note that a *restore attribute* is not permitted with the *nocopy attribute*.

In addition to the explicit distributions described above, a formal parameter 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 subroutine entry. The set of all possible argument distributions can be specified by an optional *distribution range attribute* as used in the dynamic distribution declaration. A formal parameter which inherits its distribution may not be a target of a distribute statement even if the corresponding actual argument is dynamic.

Thus a number of different situations may arise in conjunction with the transfer of distributed arrays to subroutines. 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 subroutine, array B must be redistributed: since B is static, the original distribution must be restored when the subroutine is exited. The dynamic array C is the actual argument of the (static) formal parameter $C1$: 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. The local parameter $D1$ is distributed in the same way as the formal parameter $A1$, and will thus have a cyclic distribution in this incarnation of the subroutine. Local array $E1$, in contrast, will always have a block distribution.

2.4 Accessing Distributions

The function **DISTRIBUTION**, used for alignment earlier, can also be used to inquire about the distribution of an array during the execution of the program. This may be used, for example, to determine the current distribution of a dynamically distributed array or that of a formal parameter which may inherit several different distributions as described in the last subsection. In general the values of two distributions can be compared, using the new relational operator "==" as shown below:

IF DISTRIBUTION(A) == (**BLOCK**, **CYCLIC**(2)) **THEN**
...
ENDIF

The operator "==" can also be used for pattern matching as shown here

IF DISTRIBUTION(A) == (**BLOCK**, *) **THEN**
...
ENDIF

where the match is successful *iff* the distribution of A consists of two components and the evaluation of the first component yields block distribution. The * denotes a "don't care" condition, i.e, in the above example, (**BLOCK**, **BLOCK**), (**BLOCK**, :) and (**BLOCK**, **CYCLIC(2)**) would match while (**CYCLIC**, **BLOCK**) and (**BLOCK**, **BLOCK**, **BLOCK**) would not match.

Finally, pattern matching can be combined with a side effect, s shown in the code fragment below:

IF DISTRIBUTION(A) == (**BLOCK**, **CYCLIC**(?K)) **THEN**
...
ENDIF

Here, the match is successful *iff* the distribution of A yields (**BLOCK**,**CYCLIC**(L)), where L may be any integer number greater than 0. In addition, the value L of the block length is assigned to the variable K.

The language also provides a distribution case statement, the **SELECT DCASE** statement, which can be used to execute different pieces of code based on the distribution of an array variable. The **SELECT DCASE** statement, as shown below, is modeled on the FORTRAN-90 case construct.

```
REAL A(100)  DYNAMIC ( BLOCK )
 ...
SELECT DCASE (=A)
   CASE ( BLOCK)
     ...
   CASE ( CYCLIC(?K))
     ...
   CASE ( DEFAULT)
     ...
END SELECT
```

The different code segments can, thus, be written (and optimized) based on the actual distribution of A.

2.5 Advanced Features

In the above subsections, we have provided an overview of the basic features of Vienna Fortran. A large percentage of scientific codes can be expressed using these constructs for efficient mapping onto distributed memory architectures. However, the full language provides more extensive and general mechanisms for situations where the basic features may not be sufficient.

In particular, the extended language, as described in [27], allows a more general mapping of data arrays onto arbitrary subsets of processors. For example, skewed distributions cannot be described based on the above set of features, but may be specified in the full language. It is also possible to mix and match distribution and replication such that a data array may be distributed along one dimension of the processor array while being replicated across another dimension.

In the extended language, the user can specify general alignment functions. This enables the precise specification of a mapping between the index sets of two arrays, so that even complex alignments may be described.

Other features which have not been described here but are part of Vienna Fortran include parallel loops, *on clauses* for mapping of code to processors, and input/output constructs for efficient use of secondary storage.

3 Examples

In this section, we show how the basic features described in the last section 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 Vienna Fortran.

3.1 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. [24])).

Hence, as shown in Figure 1, the second dimension of A is cyclically distributed across the processors available at load time. Note that the processor declaration is, in this case, optional since this is also the default processor structure. 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). Full details of the concurrent file operations can be found in [27].

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.** if any of the elements of its argument are true. We do not reproduce *ANY* here; it is an intrinsic function in Fortran 90. 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, 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

```
      PROGRAM Gauss

      PARAMETER (N = 4000)
      PROCESSOR (P)

      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
      ENDIF

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

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.

3.2 ADI Iteration

ADI (Alternating Direction Implicit) is a well known and effective method for solving partial differential equations in two or more dimensions [14]. 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, **$PL**.

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

      PROCESSORS (P)

      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 and 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.

3.3 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.

```
      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     & 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

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 [10, 22] are needed to generate and exploit the communication pattern.

4 Related Work

The increasing importance of distributed memory multiprocessing computers and the difficulty of programming them has led to a considerable amount of research in several related areas.

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 [17]). Languages for coordinating individual threads of a parallel program, such as LINDA [1] and STRAND [4], 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.

We discuss below just a few of the developments in language design and compiler technology which are related to Vienna Fortran, in which many of the tasks usually associated with explicit user parallelization of code are expected to be performed by the compiler. Some of the efforts mentioned below are described in articles included in this book.

The concept of processor arrays and distributing data across such arrays was first introduced in the programming language BLAZE [11] in the context of non-uniform access time shared memory machines. The Kali programming language [15], for distributed memory machines, an outgrowth of the BLAZE effort, requires the specification of data distribution in much the same way that Vienna Fortran does. Both standard and user-defined distributions are permitted; a forall statement allow 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. Again, the programmer does not specify the communication explicitly, but must mark non-local accesses.

The programming language Fortran D [5], 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. 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.

Griswold et al. [7], take a different approach introducing the concept of *ensembles* to partition data, code and communication ports. The data is partitioned into sections, each section being mapped to a processor. This allows the data to scale with the number of processors as is the case with our system. However, the communication graph and the actual movement of data, as well as the code for each processor, has to be explicitly specified by the programmer.

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 [28]. 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) [26] 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 [10] 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 [16] and ASPAR [9] (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 [19], Crystal [13] have also been developed which are targeted to distributed memory machines.

Quinn and Hatcher [8], and Reeves et al. [3, 18] compile languages based on SIMD semantics. These attempt to minimize the interprocessor synchronizations inherent in SIMD execution. The AL compiler [23], 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 [22]. These primitives have been integrated into a compiler [25] 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 [10, 12]. Explicit run-time generation of messages is also considered by [3, 13, 19], however, none of these groups save the extracted communication pattern to avoid recalculation.

5 Conclusions

In view of the increasing investment in 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 important 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 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 in a suitable manner. 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.

Acknowledgments

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

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] 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.

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

[5] 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.

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

[7] W. Griswold, G. Harrison, D. Notkin, and L. Snyder. Scalable abstractions for parallel programming. In *Proceedings of the Fifth Distributed Memory Computing Conference*, April 1990.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

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

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

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

[19] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Conference on Programming Language Design and Implementation*, pages 1–999. ACM SIGPLAN, June 1989.

[20] 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.

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

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

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

[25] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 26–30, 1991.

[26] H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1986.

[27] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – a language specification. In preparation, Austrian Center for Parallel Computation, University of Vienna, Vienna, Austria, 1991.

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