# DYNAMIC DATA DISTRIBUTIONS IN VIENNA FORTRAN*

*Barbara Chapman*[a] *Piyush Mehrotra*[b] *Hans Moritsch*[a]
*Hans Zima*[a]

[a]Institute for Software Technology and Parallel Systems,
University of Vienna, Brünner Strasse 72, A-1210 VIENNA AUSTRIA
E-Mail: zima@par.univie.ac.at

[b]ICASE, MS 132C, NASA Langley Research Center, Hampton VA. 23681 USA
E-Mail: pm@icase.edu

**Abstract**

Vienna Fortran is a machine-independent language extension of Fortran, which is based upon the Single-Program-Multiple-Data (SPMD) paradigm and allows the user to write programs for distributed-memory systems using global addresses. The language features focus mainly on the issue of distributing data across virtual processor structures. In this paper, we discuss those features of Vienna Fortran that allow the data distributions of arrays to change dynamically, depending on runtime conditions. We discuss the relevant language features, outline their implementation and describe how they may be used in applications.

**Keywords:** Distributed-memory multiprocessors, data parallel algorithms, dynamic data distributions.

*The work described in this paper was supported by the Austrian Research Foundation (FWF) and by the Austrian Ministry for Science and Research (BMWF). This research was also supported by the National Aeronautics and Space Administration under NASA contract NAS1-19480 while the authors were in residence at ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23681-0001. The authors assume all responsibility for the contents of the paper.

# 1 Introduction

High-level language extensions to Fortran, which enable users to design programs for massively parallel computers much as they are accustomed to on a sequential machine, have been the subject of intense discussion and research activity in recent months. Vienna Fortran [3, 4] is one of several proposals put forth for such a set of language extensions [5, 6, 9, 12, 13]. A number of features of Vienna Fortran have since been adopted by the High Performance Fortran Forum. One of these is the concept of static and dynamic distributions of arrays in a program, although the details of these features are not the same in High Performance Fortran (HPF).

The language extensions provided by Vienna Fortran allow the user to explicitly control and specify the mapping of arrays across the underlying set of processors. The computation, however, is still specified using a global address space which is independent of the distribution of the data. That is, the programmer writes code using a single thread of control just as when writing a sequential program. It is the compiler's responsibility to produce code suitable for parallel execution.

The Vienna Fortran Compilation System generates code based on the *SPMD* (Single Program Multiple Data) model, in which each processor executes essentially the same code, but on a local data set. The mapping specification provided by the user determines the *ownership of data*: a processor owns the data which is distributed to it, and stores it in its local memory. In general, the compiler distributes work based upon the *owner computes* rule: the processor performs the computation that defines data elements owned locally. The compiler satisfies any non-local references required for this computation by inserting communication statements to transfer the data.

The performance of the generated code is critically dependent on the data distribution used for the program. A distribution is selected with the aims of spreading the workload as evenly as possible across the processors, while preserving the locality of computation. The appropriate distribution for a given code will depend on the characteristics of both the program itself and that of the target architecture. The former includes factors such as the data access patterns exhibited by the code, and the size of the data structures relative to the number of processors used for a particular execution. The hardware factors include the communication latency and bandwidth, the computation/communication ratio, and the cache behavior of the machine.

If these factors can be determined statically, then the user can choose the "best" data distribution at compile time. However, in situations where the program behavior is dependent on runtime values, the choice of the appropriate distribution may be made at runtime

if there is language support for dynamic distributions. Major uses of dynamic distribution of data in programs are to:

- improve the locality of data accesses in codes with identifiable computation phases,
- write highly portable code in which the data distributions are selected on the basis of input data and/or characteristics of the executing machine,
- maintain a good load balance throughout the execution of a program for which the workload varies significantly during the computation.

There are significant costs associated with using dynamic distribution of data. At run time, this includes the cost of performing the actual data transfers and the cost of maintaining runtime information about the current distribution. At compile time, a more rigorous analysis must be performed to determine the distributions associated with a particular data reference. In particular, the compiler has to generate code which allows for the possibility that several data distributions may reach some statements. Despite these costs, the judicious use of dynamic distribution features can reduce the overall communication costs of the program while improving the load balance. Thus, the overall performance of the code may improve even in the presence of the runtime overheads.

In this paper, we present the language features of Vienna Fortran which support dynamic distribution of data. Section 2 describes the distribution facilities along with some control constructs required for expressing code in the presence of redistribution of data. The compiler and runtime support required for implementing these features is discussed in Section 3 while their usefulness for scientific codes is considered in Section 4. The paper concludes with a discussion of related work and some final remarks.

# 2 Distribution and Alignment in Vienna Fortran

The Vienna Fortran language extensions include features for the specification of the **processors** which execute the program, the **distribution** of arrays to subsets of processors, **alignment** between arrays, flexible mechanisms for the transfer of arguments to **procedures**, and explicitly parallel asynchronous **forall loops** [4, 16]. In this section, we focus only on the aspects relevant in the context of dynamic array distributions.

## 2.1 Basic Notation and Terminology

Each array $A$ is associated with an **index domain** which we denote by $\mathbf{I}^A$. An **index mapping** from an index domain $\mathbf{I}$ to an index domain $\mathbf{J}$ is a total function $\iota : \mathbf{I} \rightarrow \mathcal{P}(\mathbf{J}) - \{\phi\}$, where $\mathcal{P}(\mathbf{J})$ denotes the powerset of $\mathbf{J}$.

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

**Definition 1** *Let $A$ denote an array, and $R$ a processor array. An index mapping $\delta_R^A$ from $\mathbf{I}^A$ to $\mathbf{I}^R$ is called a* **distribution** *for $A$ with respect to $R$.*

An **alignment** establishes a relationship between elements of different arrays such that corresponding elements are guaranteed to reside in the same processor:

**Definition 2** *Let $A, B$ denote arbitrary arrays. An index mapping $\alpha_B^A$ from $\mathbf{I}^A$ to $\mathbf{I}^B$ is called an* **alignment** *for $A$ with respect to $B$.*

*Given $\delta_R^B$, $\delta_R^A$ is determined as follows: For each $\mathbf{i} \in \mathbf{I}^A$ :*

$$\delta_R^A(\mathbf{i}) := CONSTRUCT(\alpha_B^A, \delta_R^B) = \bigcup_{\mathbf{j} \in \alpha(\mathbf{i})} \delta_R^B(\mathbf{j})$$

## 2.2 Specification of Distribution and Alignment

Distributions are specified in a program by **distribution expressions**. Each distribution expression, for example *(BLOCK,CYCLIC(K))*, determines a class of distributions which is called a **distribution type**. The application of a distribution type to a (data) array and a processor section yields a distribution.

*Simple* distribution expressions specify mappings between one array dimension and one processor dimension; they include the intrinsic distribution functions *BLOCK*, *CYCLIC*, *S_BLOCK*, and *B_BLOCK*. *BLOCK* distributes one array dimension to one processor dimension in evenly sized segments. *CYCLIC* maps elements of an array dimension in a round-robin fashion to a dimension of the processor array. *S_BLOCK* and *B_BLOCK* permit the specification of contiguous irregular blocks, introducing the concept of *general block* distributions.

Distribution expressions associated with multi-dimensional arrays may be specified as a list of simple distribution expressions, each one corresponding to exactly one array dimension. The **elision symbol** ":" in such a list prevents the associated array dimension from being distributed.

Alignments are expressed in a Vienna Fortran program by **alignment specifications**. We illustrate their use – together with distribution expressions – by a simple example:

**Example 1 Distribution and Alignment**

**PARAMETER** (M=2)
**PROCESSORS** R(1:M,1:M)
**REAL** C(10,10,10) **DIST** (BLOCK,BLOCK,:) **TO** R
**REAL** D(10,10,10) **ALIGN** D(I,J,K) **WITH** C(J,I,K)

*R denotes a two-dimensional processor array. The distribution of array C is specified by the distribution expression* (BLOCK,BLOCK,:) *which indicates that the first two dimensions are distributed by* BLOCK*, while the third dimension is not distributed. More precisely,* $\delta^C(i,j,k) = \{R(\lceil \frac{i}{5} \rceil, \lceil \frac{j}{5} \rceil)\}$ *for all* $k, 1 \leq k \leq 10$*. The alignment specification for D transposes the first and second dimensions of C, i.e., the resulting alignment function maps each index triplet* $(i,j,k)$ *in* $\mathbf{I}^D$ *to the index triplet* $(j,i,k)$ *in* $\mathbf{I}^C$*.*

## 2.3 Dynamically Distributed Arrays

The language distinguishes between statically and dynamically distributed arrays, depending on whether or not the association between an array and its distribution is invariant in a given scope*. This distinction is made syntactically in the declaration of the array. The arrays shown in Example 1 were statically distributed.

We define an equivalence relation, **connect**, in the set of dynamically distributed arrays within a given scope. This relation satisfies the following conditions:

1. Each equivalence class consists of one distinguished member, the **primary array**, $B$, of the class, and 0 or more **secondary arrays**. We denote the class associated with primary array $B$ by $\mathcal{C}(B)$.

2. The distribution of each secondary array $A \in \mathcal{C}(B)$, if any, is defined in the declaration of $A$ by referring to $B$ in a *secondary array annotation*, which specifies a *connection* by distribution extraction [16] or alignment.

3. Distribute statements are explicitly applied to primary arrays only; their effect is to redistribute all arrays in the associated equivalence class so that the *connection* is maintained.

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

5. The connect relation does not extend across procedure boundaries.

*If no ambiguity is possible, we simply refer to *static* or *dynamic* distributions.

An annotation specifying $B_1, \ldots, B_r$ as *primary arrays* has the form

**REAL** $B_1(\ldots), B_2(\ldots), \ldots, B_r(\ldots)$ **DYNAMIC** [,*distribution-range*] [,*initial-distribution*]

A **distribution range** determines the set of all distribution types (or a superset thereof) which can be associated with the arrays $B_i$ during the execution of the procedure in which the declaration occurs. The distribution range is specified by the keyword **RANGE**, followed by a parenthesized list of *distribution expressions* (see Section 2.2). The "*" can be used as a "don't care" symbol. Distribute statements applied to the $B_i$ must respect the restrictions imposed by this attribute.

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

An **initial distribution** is evaluated and associated with each $B_i$ each time the array is allocated. An array for which an initial distribution has not been specified cannot be legally accessed before it has been explicitly associated with a distribution by the execution of either a distribute statement or a procedure call.

A *secondary array annotation*, for the arrays $A_1, \ldots, A_s$, has the form

**REAL** $A_1(\ldots), \ldots, A_s(\ldots)$ **DYNAMIC**, **CONNECT** *connection*

The *connection* can be either a *distribution extraction* [16], or an *alignment specification*. In both cases, all secondary arrays $A_j$ are **connected** to a primary array $B$. As a result of this declaration, the $A_j$ are entered into the equivalence class $\mathcal{C}(B)$.

**Example 2 Dynamic array annotations**

```
  REAL B1(M) DYNAMIC
  REAL B2(N) DYNAMIC, DIST ( BLOCK )
  REAL B3(N,N), B4(N,N) DYNAMIC, RANGE (( BLOCK,BLOCK),(*,CYCLIC) ),
&           DIST ( BLOCK,CYCLIC )

  REAL A1(N,N) DYNAMIC, CONNECT (=B4)
 REAL A2(N,N) DYNAMIC, CONNECT A2(I,J) WITH B4(I,J)
```

*All arrays declared here are dynamically distributed; $B1$ through $B4$ are primary, $A1$ and $A2$ secondary arrays. For $B1$, no distribution range and no initial distribution are given. For $B2$, no distribution range is given, and $(BLOCK)$ is specified as initial distribution. For $B3$ and $B4$, a distribution range as well as an initial distribution are specified. $A1$ is connected to $B4$ via distribution extraction while $A2$ uses an (identity) alignment to specify the connection. As a consequence, $\mathcal{C}(B4) \supseteq \{B4, A1, A2\}$; the connections specified ensure that the distribution type of $A1$ and $A2$ will be always the same as that of $B4$.*

## 2.4 Distribute Statements

A *distribute-statement* has the form

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

where $B$ is an array name associated with a primary array, and *da* is either a *distribution expression*, possibly associated with a processor section, or an *alignment specification.*

The distribute statement is executed as follows:
First, a set *NOTRANSFER* is determined as the set of all names specified in the *notransfer-attribute*, or the empty set in the default case. All names in
*NOTRANSFER* must be secondary arrays in $\mathcal{C}(B)$.
Secondly, *da* is evaluated; its result is used to determine a distribution, $\delta^B$, for array $B$.
Thirdly, for each secondary array $A$ in $\mathcal{C}(B)$, its distribution, $\delta^A$, is determined from the distribution type associated with *da*, $\mathbf{I}^A$, and the *connection* between $A$ and $B$, as established in the associated secondary array annotation. If $A$ is a member of *NOTRANSFER*, then only the access function for $A$ is changed and the elements of the array are not physically moved.

**Example 3 Distribute Statement**

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

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

*In the first statement, the array* $B1$ *is distributed by* $(BLOCK)$.

*In the second statement,* $B1$ *and* $B2$ *(both of which are currently distributed by* $(BLOCK)$*) are redistributed as* $(CYCLIC(k'))$*, where* $k'$ *denotes the value assigned to the variable* $K$ *in the assignment* $K = expr$.

*The third statement redistributes* $B3$ *as* $(BLOCK, CYCLIC)$*; in the next statement,* $B4$ *and the associated secondary arrays* $A1$ *and* $A2$ *are distributed as* $(CYCLIC(k'), CYCLIC(3))$.

## 2.5 Control Constructs

The capability to redistribute data at an arbitrary position in a Vienna Fortran program, including within conditionals, implies that

- an array reference in the program may, at run-time, be reached by more than one distribution for the array, and
- the compiler may not be able to determine precisely the set of all distributions reaching such a reference, no matter how much analysis is performed.

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

### 2.5.1 The DCASE Construct

The *dcase-construct* has the form

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

where

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

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

1. The distribution of each selector, and its type, are determined.

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

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

A condition $c_j$ **matches** iff either $c_j$ is the keyword **DEFAULT** , or $c_j$ is a list of queries, each of which matches. Each query tests the distribution of one selector array. Query lists may be either **positional** or **name-tagged**. In a positional query list, the queries are associated with the selectors $A_1, A_2, \ldots$ in this order. In a name-tagged query list, the selector associated with each query is explicitly specified by a *name-tag*. The order in which the queries occur in such a list is semantically irrelevant. A query list need not contain a query for every selector. In such a case, an implicit "*" is inserted for every selector which is not represented.

Full details of the matching process are given in [16].

**Example 4 The dcase construct**

```
  REAL B1(M) DYNAMIC
  REAL B2(N) DYNAMIC, DIST ( BLOCK )
  REAL B3(N,N), DYNAMIC, RANGE (( BLOCK,BLOCK ),
&      (CYCLIC, CYCLIC(*)),(*,CYCLIC) ), DIST ( BLOCK,CYCLIC )
          ...
  SELECT DCASE (B1,B2,B3)
   CASE ( BLOCK),(BLOCK),(CYCLIC(2),CYCLIC)
            a1
   CASE B1: ( CYCLIC ), B3:( BLOCK,* ))
            a2
   CASE B3:( BLOCK,CYCLIC )
            a3
   CASE DEFAULT
            a4
  END SELECT
```

*In the following, let $t_i$ denote the distribution type associated with $B_i$.*

*The first query list is positional; it matches if $t_1 = t_2 = (BLOCK)$, and $t_3 = (CYCLIC(2), CYCLIC)$.*

*The second list is name-tagged; it matches if $t_1 = (CYCLIC)$, $t_3 = (BLOCK, t')$, where $t'$ is arbitrary, and $t_2$ is any distribution type.*

*The third query list matches if* $t_3$ = (BLOCK, CYCLIC). $t_1, t_2$ *are irrelevant in this case.*

*Finally, the fourth query list is always matched. Thus, if none of the first four query lists match, then* $a_4$ *will be executed.*

### 2.5.2 The IF Construct

The *if-construct* of Vienna Fortran is based upon a **generalized logical expression**, which is a Fortran *logical_expression* that in addition may contain references to the intrinsic function *IDT*. This function performs a test of the distribution types associated with their arguments and, optionally, of the processor sections to which the arguments are distributed; it yields a logical value. For example, the second clause in the *dcase construct* above can be explicitly expressed as

$$\textbf{IF } (IDT(\text{B1},(CYCLIC))) \textbf{ .AND. } (IDT(\text{B3},(BLOCK(*)))) \textbf{ THEN}$$
$$a_2$$

# 3 Implementation

In this section, we briefly describe, at an abstract level, the support for dynamic data distributions in the Vienna Fortran Compiler System (VFCS). More details of the compilation strategy used in VFCS are given in [7, 17]. Some of the issues discussed here are also being handled in other systems [1, 2, 8, 14].

The features required to manage dynamic data distribution comprise both *compile time* and *run time* elements. Most of these features are actually required to handle other aspects of Vienna Fortran: in particular, many of the problems posed by run time redistribution of data structures are the same as, or similar to, those posed by the redistribution of arrays at subroutine boundaries, and those posed by the fact that in any code, several arrays, with possibly distinct distributions, may be bound to the same formal argument of a subroutine.

## 3.1 Compiler Support

There are two major phases in the compiler: analysis and code generation. The most important task in the analysis phase is solving the reaching distribution problem: that is, the compiler must determine the *range* of distribution types which may reach a specific array access in the code, by intra- and inter-procedural analysis. This is performed both for declared (and explicitly distributed) arrays as well as for formal subroutine arguments. The system constructs pairs consisting of a distribution type and a target processor array. We call the set of all such pairs which is valid for a specific array at a specific position in the

program the set of *plausible distributions*. The information computed may not be precise, since some of the distributions may not actually be assumed at run time. If the full code is not available, the compiler will have to rely on range specifications provided by the user, or make worst case assumptions.

An extensive communication analysis provides not only information on the communication associated with each plausible distribution for an array, but also the memory requirements of the array under that distribution. The details of this analysis are outside the scope of this paper.

The compiler also performs a partial evaluation of distribution queries (both IDT and the dcase construct), by checking whether there is a plausible distribution which will match.

The compiler must perform many related tasks during during code generation. In particular, it generates code to create and maintain data structures describing the distributions and other attributes of arrays, such as the associated overlap areas. The compiler also inserts calls to run time routines to perform communication as necessary and to routines which perform the redistribution of data.

## 3.2 Run Time Support

The **run time** support required may be described as the **Vienna Fortran Engine (VFE)**, an *abstract machine* that executes Vienna Fortran object programs. VFE is a machine at a higher level of abstraction than the vendor-supplied hardware/operating system interface. It is realized by a set of run time libraries which provide the required functionality on a specific target architecture. In particular, these provide complex data organization and access schemes, and high-level operations:

- The **memory management scheme** of the VFE is inherently dynamic. Even without dynamic distributions, the actual allocation of an array to the processors' memories may not be known. Redistribution requires, in addition, the possibility of reallocation.
- The **data organization and access features** provided by the VFE include:

  1. Data access functions for Vienna Fortran distributions (including the implementation of irregular accesses via translation tables and sophisticated buffering schemes for accesses to non-local objects, as implemented in the PARTI routines [15]).
  2. An interface for external distribution generators and specifiers.
  3. Run time optimization of communication related to dynamic array references.

- A run time library of communication routines for transferring single array elements and array sections, including specialized routines for handling reductions.
- Routines to perform the tasks associated with **DISTRIBUTE**, construct access functions, to modify descriptors associated with arrays (this information may be modified when the distribution is changed, or on entry to a subroutine), and test information stored in these as required for the implementation of IDT and the dcase construct.

### 3.2.1 Run-Time Representation of Arrays

Some of the relevant components of the information related to an array stored locally in each processor are the data structures and access functions listed below. Here, $A$ denotes an array name, and $p$ a processor.

**Data Structures:**

- *index_dom(A)* specifies the *index domain* of $A$.
- *dist*($A$) characterizes the distribution of $A$, which includes a distribution type, and a specification of the target processors. For certain complex distributions, a pointer to a translation table is required[†].
- *connect_class*($A$) determines the set of secondary arrays *connected* to a primary array.
- *alignment(C)* specifies, for each array $C$ in *connect_class(A)*, the alignment of $C$ with respect to $A$.
- For every $\mathbf{i}$ such that $A(\mathbf{i})$ is owned by processor $p$, $loc\_map_p^A(\mathbf{i})$ specifies the offset of $A(\mathbf{i})$ in the local memory of processor $p$.
- For regular and irregular BLOCK distributions, $segment_p^A$ specifies the sequence of the local lower and upper bounds in each dimension.

**Access Functions**

- Access in processor $p$ to *local* array element $A(\mathbf{i})$ is performed by evaluating $loc\_map_p^A(\mathbf{i})$.
- Access in processor $p$ to a *non-local* array element $A(\mathbf{i})$ is performed by determining a processor $q$ owning $A(\mathbf{i})$ from *dist(A)*, and inserting message passing operations that send the required element from $q$ to $p$.

[†]For dummy arguments, the description may include a pointer to another array representation and/or sectioning operations.

### 3.2.2 Implementation of DISTRIBUTE

Consider the statement

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

where *da* is a distribution expression or an alignment specification, and the *notransfer-attribute* determines the set $NOTRANSFER=\{C_1, \ldots, C_m\}$ (see Section 2.4). The realization of this statement is handled by a run-time routine executed on each processor which is passed the array and its current set of descriptors and returns new descriptors. Each processor determines the new locations of current local data, sends it to the new locations, and receives data from other processors. Data motion is suppressed where data flow analysis, or a NOTRANSFER specification, permits.

This corresponds to executing the following sequence of steps on each processor:

- **Step 1: Evaluate the new distribution and the associated access functions**

  1. Evaluate the new distribution: *dist(B):= eval(da)*
  2. Determine the functions *loc_map* and *segment* from *dist(B )*

- **Step 2: Determine the distributions of the arrays** *connected* **to** $B$**:**

  **for every** $C \in connect\_class(B) - \{B\}$ **do**
  $dist(C) := CONSTRUCT(alignment(C), \delta^B)$
  **endfor**

  Here, the application of the function *CONSTRUCT* to the alignment function associated with $C$ and the new distribution of $B$ yields the new distribution of $C$.

- **Step 3: Communicate**

  **for every** $C$ **such that** $(C \in connect\_class(B) - NOTRANSFER) \wedge$ (the previous distribution of $C$ is *old_dist(C)*) **do**
  *COMMUNICATE(C,old_dist(C),dist(C))*
  **endfor**

```
      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 J = 1, NY
        CALL TRIDIAG( V(:, J), NX)
      ENDDO

      DISTRIBUTE V :: ( BLOCK, : )

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

Figure 1: ADI iteration in Vienna Fortran

# 4 Applications

In this section, we discuss the benefits of dynamic distribution of data for scientific codes. We present several examples in which using dynamic data distributions allows the user to choose the appropriate data distribution based on the runtime behavior of the program.

Consider first the case in which a runtime value determines the choice of the best distribution. For example, in a grid based computation, such as smoothing, the value at a grid point is based on its 4 nearest neighbors. A column distribution of the $N \times N$ grid will give rise to 2 messages per processor, each of size $N$, per computation step. On the other hand, if the grid is distributed by blocks in two dimensions across a $p^2$ processor array, then each computation step requires 4 messages of size $N/p$ each on each processor. Thus, given the startup overhead and cost per byte of each message of the target machine, the ratio $N/p$ will determine the most appropriate distribution. If the code has been written such that the size of the grid is an input parameter, then the user can use the dynamic distribution facilities of Vienna Fortran to set the distribution of the grid[‡].

Another class of codes which can benefit from dynamic distributions are codes which exhibit different data access patterns in different phases of the program. Dynamic data

[‡]Vienna Fortran supports an intrinsic function **$NP** which returns the number of processors being used to execute the program and can be used to compute the ratio $N/p$.

distributions can be used to control the locality of data access in such codes. For example, consider ADI (Alternating Direction Implicit) codes [11] used for solving partial differential equations 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. The tridiagonal solve has a recurrence and thus generates data dependencies along the columns in the first phase and along the rows in the second phase.

There are two broad choices in such situations [4]. We could choose a single distribution for the whole program so that data accesses are satisfied locally in one phase while paying the communication costs in the other phase. On the other hand, we could dynamically redistribute the data so that data accesses in all phases are satisfied locally.

In Figure 1, we present a Vienna Fortran code fragment which employs the latter strategy. The tridiagonal solves are performed by a sequential routine *TRIDIAG* (not shown here) which is given a right hand side and overwrites it with the solution of a constant coefficient tridiagonal system. The array $V$ is declared as **DYNAMIC** and is initially distributed by block in the second dimension. Thus, in the first loop which performs the sweep over columns (representing x-lines), each column is local to a processor and causes no communication. The array is then explicitly remapped to be distributed by block in the first dimension. This allows the second loop, a sweep over y-lines, to also be executed without any communication. Thus, all the communication is confined to the redistribution operation, with only local accesses during the computation.

If the array is not explicitly redistributed between the two loops, then the argument to the second call to *TRIDIAG* is distributed across a set of processors and it becomes the responsibility of the compiler to embed the required communication in the generated code. The efficiency of the resulting code will depend on various factors including, in particular, the analysis capabilities of the compiler. The dynamic distribution facilities of Vienna Fortran make it easy for the user to restrict the communication to the redistribution operation which, at least in the above code, can be implemented by an efficient pre-compiled routine.

For the examples given above it is possible to write the code without using explicit redistribution statements. For example, one could declare two or more arrays with different static distribution and use array assignments to produce the effect of redistribution. This approach, clearly, wastes storage space since only one of the arrays would be fruitfully used in any single computation phase.

Another approach is to use procedure boundaries for implicit redistribution of data. Vienna Fortran allows procedure arguments to be declared with a specific distribution. When the procedure is called, it is the compiler's responsibility to redistribute the actual argument to match the specified distribution. Thus, the ADI example could be rewritten such that it calls a different subroutine in the second loop, one which specifically declares its argument to be distributed by block in the first dimension. Similarly, the grid example could be written such that a different subroutine is called, depending on the ratio of the size of the grid and the number of executing processors. The problem, however, is that this approach may lead to an explosion of subroutines which are different only in the distribution specified for their arguments.

Another problem with using either assignment or procedure boundaries for implicit redistribution is that the approaches are particularly awkward and cumbersome to use if there is an outer iterative loop around the phases requiring redistribution. Further, it is not always feasible to write a program such that distributions change only at procedure boundaries. For example, in applications such as adaptive mesh codes or particle-in-cell (PIC) codes, the work distribution changes as the computation progresses. In such codes, the data needs to be redistributed dynamically in order to rebalance the workload.

Consider a simulation code based on the particle-in-cell method, which can be used to study the motion of particles in a given domain, such as plasmas for controlled nuclear fusion, or stars and galaxies. The computation at each time step can be divided into two phases. In the first phase, a global force field is computed using the current position of particles. In the second phase, given the new global force field, new positions of the particles are computed. The program can be structured by dividing the underlying domain into cells with each cell owning a set of particles. The particles move from one cell to another as they change positions across the domain. Since the computation in each cell is dependent on the number of particles in the cell, the workload across the domain changes as the computation progresses.

Figure 2 shows the outermost level of a simplified version of a PIC code as expressed in Vienna Fortran. The code omits details irrelevant to the discussion here. In this code, the cells are represented by the first dimension of the array *FIELD*. There are a maximum of *NCELL* cells and each cell is restricted to have a maximum of *NPART* particles.

The main goal here is to distribute the cells across the processors such that the work per processor is approximately equal. In this code, we use the generalized block distribution to distribute the cells in irregular (but contiguous) blocks to the processors. The block sizes (i.e., the number of contiguous cells) are selected so that each processor has roughly the same number of particles on its local part of the domain.

```
      PARAMETER (NCELL = ..., NPART = ...)

      INTEGER BOUNDS($NP)
      REAL FIELD(NCELL, NPART, ...) DYNAMIC, DIST ( BLOCK, :, :)

C     Compute initial position of particles
      CALL initpos(FIELD, NCELL, NPART, ...)

C     Compute initial partition of cells
      CALL balance(BOUNDS, FIELD, NCELL, NPART, ...)
      DISTRIBUTE FIELD :: B_BLOCK (BOUNDS)

      DO k = 1, MAX_TIME
C       Compute new field
        CALL update_field(FIELD, NCELL, NPART, ...)
C       Compute new particle positions and reassign them
        CALL update_part(FIELD, NCELL, NPART, ...)

C       Rebalance every 10th iteration if necessary
        IF ( MOD(k,10) .EQ. 0 .AND. rebalance() ) THEN
          CALL balance(BOUNDS, FIELD, NCELL, NPART, ...)
          DISTRIBUTE FIELD :: B_BLOCK (BOUNDS)
        ENDIF

      ENDDO
```

Figure 2: High level PIC code in Vienna Fortran

The array *FIELD* is declared to be **DYNAMIC** with the first dimension initially distributed into regular blocks. The procedure *initpos* determines the initial position of the particles and places them in the appropriate cells. Using the number of particles in each cell, the procedure *balance* computes the block sizes to be assigned to each processor. It stores these in the array *BOUNDS*, which is then used to redistribute the array *FIELD* via the the intrinsic distribution function *B_BLOCK)*.

In each time step (represented by one iteration of the outer loop), the procedure *update_field* computes the new force field based on the current particle positions. Then, the procedure *update_part* is called to update the positions of the particles. Based on the new positions, the new owner cell for each particle is determined. If a particle has moved from one cell to another, it is explicitly reassigned. This obviously requires communication if the new cell is on a different processor. Since this communication is based on the locations of the current and the new cell, it is highly irregular in nature. Thus, the compiler will have to generate runtime code using the inspector/executor paradigm [10, 15] to support this particle motion.

If the number of particles on each processor remains roughly equal for the duration of the simulation, then load balance will be maintained. Some problems of this kind display sufficient uniformity such that a simple block distribution will suffice to provide a reasonable load balance. For other problems, the motion of particles during the simulation may lead to a severe load imbalance. The code, as shown here, checks on every 10th iteration (by calling function *rebalance*) whether rebalancing is required. If so, a new *BOUNDS* array is computed and the cells redistributed to balance the workload.

The redistribution needed for such load balancing is based on the current values of some data structure, for example, in the above case it is based on the number of particles per cell. Thus, this kind of redistribution cannot be expressed using either array assignment or procedure boundaries and requires language support for dynamic distributions.

# 5 Related Work

Kali [12] was the first language to introduce dynamic data distribution in a data parallel language aimed at distributed memory machines. It provided indirect mapping and user defined distribution functions which could depend on runtime values. A distribute statement allowed the user to dynamically change the distribution of an array at runtime. The design of Kali has greatly influenced the development of Vienna Fortran.

The DINO language, which extends C by constructs for specifying virtual processors to which data may be mapped, and whose compiler is targeted to distributed memory computers, supports redistribution of data at procedure boundaries, but does not extend these mechanisms to handle other forms of user-specified run-time distribution ([14]).

An executable *DISTRIBUTE* statement which performed run-time redistribution of arrays was formulated by Marc Baber and implemented in his *Hypertasking* compiler for block distributions of arrays; the system attempted to optimize the communication required for redistribution. This system did not permit procedure calls with distributed data. It has been implemented on the Intel iPSC hypercubes [2].

The Fortran D language proposal [6] suggests a set of features for enabling the portable specification of code to run on a variety of parallel architectures, including a dynamic *DISTRIBUTE* statement. Fortran D does not, however, provide a means for static distribution of arrays, and does not include any additional constructs which might enable the user to control or structure the use of dynamic distributions. As far as we are aware, the Fortran D implementation does not yet provide for dynamic data distributions.

The High Performance Fortran proposal [9] includes static and dynamic distributions in much the same way that Vienna Fortran does and has included a small set of distribution

queries in the language constructs. It has *REALIGN* and *REDISTRIBUTE* directives to permit independent redistribution and realignment of arrays during execution. These are both subsumed by the *DISTRIBUTE* statement in Vienna Fortran. In contrast to Vienna Fortran, if an array is redistributed in a procedure, HPF does not permit the new distribution to be returned to the calling procedure.

# 6 Conclusions

Dynamic data distributions are essential for a variety of real applications, which are characterized by large variations in the size or structure of input data sets, the need to perform dynamic load balancing, or the necessity to execute the code on several different architectures or different configurations of one machine. In all these cases, the decision on how to map the data arrays to the executing processors might have to be deferred until run time.

However, the deferment of such decisions makes it difficult for the compiler to generate efficient code. This problem can be alleviated by a combination of enhanced language support, extensive intra- and inter-procedural compiler analysis, and careful structuring of the program by the user so that in all critical code sections the distribution is known at compile time.

# References


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

[2] Marc Baber. Hypertasking support for dynamically redistributable and resizeable arrays on the iPSC. In *Proceedings of the Fifth Distributed Memory Computing Conference*, 59-66, 1990.

[3] S. Benkner, B. Chapman, and H. Zima. Vienna Fortran 90. In *Proceedings of the SHPCC Conference 1992*, 51–59, April 1992.

[4] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran *Scientific Programming* 1(1):31-50, Fall 1992.

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

[6] 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.

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

[8] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[9] High Performance FORTRAN Language Specification. Technical report, Rice University, May 1993.

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

[12] 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*, pp. 364–384. Pitman/MIT-Press, 1991.

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

[14] 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.

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

[16] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – a language specification. ICASE Internal Report 21, ICASE, Hampton, VA, 1992.

[17] H. Zima and B. Chapman. Compiling for Distributed Memory Systems. *Proceedings of the IEEE, Special Section on Languages and Compilers for Parallel Machines*, February 1993.