NASA Contractor Report 191451

ICASE Report No. 93-17

ICASE

HIGH PERFORMANCE FORTRAN WITHOUT TEMPLATES: AN ALTERNATIVE MODEL FOR DISTRIBUTION AND ALIGNMENT

Barbara Chapman
Piyush Mehrotra
Hans Zima

NASA Contract No. NAS1-19480
April 1993

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23681-0001

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-0001

High Performance Fortran Without Templates: An Alternative Model for Distribution and Alignment*

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

Department for Software Technology and Parallel Systems,
University of Vienna,
Bruennerstrasse 72, A-1210 Vienna, Austria
zima@par.univie.ac.at
[b]ICASE, MS 132C, NASA Langley Research Center,
Hampton VA 23681 USA
pm@icase.edu

Abstract

Language extensions of Fortran are being developed which permit the user to map data structures to the individual processors of distributed memory machines. These languages allow a programming style in which global data references are used. Current efforts are focussed on designing a common basis for such languages, the result of which is known as High Performance Fortran (HPF). One of the central debates in the HPF effort revolves around the concept of templates, introduced as an abstract index space to which data could be aligned. In this paper, we present a model for the mapping of data which provides the functionality of High Performance Fortran distributions without the use of templates.

*Research 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, and also by the Austrian Research Foundation (FWF) and the Austrian Ministry for Science and Research. This paper is partially based on Chapter 3 of the Version 0.2 draft HPF specification [8].

1 Introduction

Much current research activity is concentrated on providing suitable programming tools for distributed-memory architectures. One focus is on the provision of appropriate high-level language constructs to enable users to design programs in much the same way as they are accustomed to on a sequential machine. Several proposals have been put forth in recent months for a set of language extensions to achieve this [3, 4, 5, 6, 10], in particular (but not only) for Fortran.

Recently, a coalition of researchers from industry, government labs and academia formed the High Performance Fortran Forum to develop a standard set of extensions for Fortran 90 which would provide a portable interface to a wide variety of parallel architectures. The forum has produced a draft proposal for a language, called High Performance Fortran (HPF), which focuses mainly on issues of distributing data across the memories of a distributed memory multiprocessor.

High Performance Fortran (HPF) adds directives to Fortran 90 to allow the user to advise the compiler on the allocation of data objects to processor memories. The three basic elements of the model are:

- **abstract processors**,
- **distributions**, which are mappings of objects to abstract processors,
- **alignments**, which are mappings of data objects to other objects.

The **distribution** of an object (usually an array) specifies a mapping of the index domain associated with the object to the index domain of a set of abstract processors. This may be specified by the user: a) **directly**, by explicitly specifying suitable directives, or b) **indirectly**, by using an **alignment** that relates the index domain of the array to the index domain of another object whose distribution is known.

The HPF directives provide a way to direct the compiler to ensure that certain data objects will reside in the same processor. The underlying motivation is that an operation on two or more data objects is likely to be carried out much faster if they all reside in the same processor, and, furthermore, it may be possible to carry out several such operations concurrently if they can be performed on different processors.

Alignment can serve as a *bundling mechanism*: once many arrays are aligned to the same object, then they can be distributed onto a processor arrangement with a single statement.

In general, arrays are aligned to other arrays. However, HPF has introduced the concept of templates to be used as an alignment base. As stated in the HPF language specification [8]:

Sometimes it is desirable to consider a large index space with which several smaller arrays are to be aligned, but not to declare any array that spans the entire index space. HPF provides the notion of a TEMPLATE, which is like an array whose elements have no content and therefore occupy no storage; it is merely an abstract index space that can be distributed and with which arrays may be aligned.

The problem with this approach is that even though it is useful in some special situations, the concept of templates necessarily complicates the whole underlying semantic model. Since templates are not first class objects in the language (they can occur only in directives), they cannot be passed across procedure boundaries, and thus cannot be used to describe the distributions and alignments of procedure arguments. Also, as currently defined, the size of templates has to be a specification expression and hence templates cannot be used for describing the alignment of Fortran 90 allocatable arrays.

In this paper, we show that the HPF distribution and alignment model can be defined in a clear and concise manner without templates, while retaining the intended functionality.

The major differences between the current HPF draft [8] and the language proposed in this paper are as follows. The model has been **simplified** by:

1. Removing template directives.
2. Limiting the height of alignment trees to 1.
3. Clarifying the role of processors by establishing a language defined mapping to an implementation-specific abstract processors arrangement.
4. Passing of arguments to procedures has been simplified by eliminating the INHERIT attribute, matching alignments, and the TO-clause for dummy arguments.

At the same time, the language has been significantly **generalized** with the objective of improving object program performance. In particular:

1. Arrays may be distributed to processor sections.
2. The set of distribution functions has been extended by including GENERAL_BLOCK. This allows the specification of irregular block distributions, which are important for the support of load balancing, and can be implemented efficiently [13].
3. The concept of *distribution functions* has been defined in a general way so that future language standards may easily incorporate more general mappings.

The paper is organized as follows. In the next section we describe the model and terminology underlying our proposal. The subsequent sections introduce the main language extensions – processors, distribution directives and alignment directives. Issues involving allocatable arrays and procedures are treated separately. We then discuss the issues arising due to HPF templates and conclude with a discussion of related work.

2 Model

2.1 Index Domains

An **index domain I of rank (dimension)** n is an ordered set of subscript tuples that can be represented by a subscript-triplet-list of length n (see Fortran 90 specification, R619). Each element of an index domain is called an **index**; it represents an n-dimensional arrangement of values. **I** is called a **standard index domain** iff the stride in each subscript triplet is 1.

Let A denote a declared data array (or processor array) that has been created. Then A is associated with a standard index domain which we denote by $\mathbf{I}^A$.

2.2 Distributions

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

Definition 2 Distributions
Let A denote an array, and R a processor array. An index mapping δ_R^A *from* $\mathbf{I}^A$ *to* $\mathbf{I}^R$ *is called a* **distribution function** *for A with respect to R.*

A distribution function δ_R^A – which is a mapping between index domains – induces an associated **element-based distribution** that maps elements of A to one or more abstract processors.*

Note that scalars can easily be accommodated in our model by treating them as if they were associated with an index domain consisting of exactly one element.

*Note that replication can be modeled as a special case of distribution, since every array element can be distributed to an arbitrary (positive) number of processors.

2.3 Alignment

Definition 3 *Let A, B denote arbitrary arrays. An index mapping α_B^A from $\mathbf{I}^A$ to $\mathbf{I}^B$ is called an* **alignment function** *for A with respect to B.*

Definition 4 Construction of a distribution
If A, B, δ_R^B, and $\alpha_B^A : \mathbf{I}^A \rightarrow \mathcal{P}\ (\mathbf{I}^B) - \{\phi\}$ are given as above, then δ_R^A can be determined as follows: For each $\mathbf{i} \in \mathbf{I}^A$:

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

We will express this relationship below in the form

$$\delta_R^A = CONSTRUCT(\alpha, \delta_R^B).$$

This can be verbally described as follows: if $\mathbf{i}$ is an index of A which is mapped to an index $\mathbf{j}$ of B via the alignment function α, then $A(\mathbf{i})$ and $B(\mathbf{j})$ are guaranteed to reside in the same processor under any given distribution for B.

2.4 The Alignment Relation

For the following discussion, we consider the data space $\mathcal{A}$ of all arrays that are accessible in a given scope, and have been created, at a given time during the execution of a program unit.

An **alignment directive** (see Section 5) establishes an **alignment** from an array A_1, the **alignee**, to an array A_2, the **alignment base**. It defines an alignment function for A_1 with respect to A_2.

An HPF program must satisfy the following constraints:

1. Each array occurring as an alignment base must not be aligned to another array. For such an array, a distribution must be specified directly.

2. Each array occurring as an alignee can be aligned with only one alignment base.

This enables us to represent $\mathcal{A}$ as an **alignment forest**, consisting of a set of **alignment trees**. The nodes in the alignment forest represent arrays, and there is a directed edge from B to A if and only if A is aligned to B. The height of alignment trees may be either 1 or 0. An alignment tree of height 0 is called **degenerate**: it consists of exactly one node that represents an array which is not aligned to any other array, and to which no other array is aligned.

Each alignment tree T has a uniquely defined root, which is called the **primary array** of T. All other nodes of T are called **secondary arrays.**

Let B denote a primary array. Then there is either a directive which explicitly specifies a distribution for B or B is implicitly distributed by the compiler. Primary arrays are the only arrays with this property.

Let A denote an arbitrary secondary array of a tree with primary array B . Then there exists an alignment function α, describing the alignment from A to B. If δ_R^B is the distribution of B, the distribution of A satisfies $\delta_R^A = CONSTRUCT(\alpha, \delta_R^B)$.

After the specification part of a unit has been completely processed, the alignment forest can be constructed for the set of all arrays that are accessible and have already been created. This is the initial state for the **actual alignment forest** associated with the processing of the executable part of the program. The structure of the forest may change dynamically during execution as a result of executing REDISTRIBUTE and REALIGN directives, ALLOCATE and DEALLOCATE statements, and procedure calls.

For the details of these manipulations see Sections 4.2, 5.2, and 7. Distribution and alignment functions are explained in Sections 4 and 5, respectively.

3 The Processors Directive

Each implementation of HPF determines uniquely an **implicit abstract processor arrangement**, AP, which specifies a linear numbering scheme for the physical processors of the underlying machine.

The PROCESSORS directive declares one or more processor arrangements, each of which may be either a **processor array arrangement** or a conceptually **scalar processor arrangement.**

The specification of a processor arrangement determines the name and, in the case of a processor array arrangement, a non-empty index domain. It must appear in the specification part of a program unit.

Each processor arrangement is mapped to AP in the same way as storage association is defined for the Fortran 90 EQUIVALENCE statement, with abstract processors playing the role of the storage units (see Fortran 90 specification, 5.5.1). The sharing of an abstract processor implies the sharing of the associated physical processor.

Depending on the target architecture, data distributed to a (conceptually) scalar processor arrangement may reside in a single control processor (if the machine has one), or may reside in an arbitrarily chosen processor, or may be replicated over all processors. The language does not specify a relationship between different scalar processor arrangements.

4 Distribution Directives

The DISTRIBUTE directive specifies the distribution (Section 2.2) of one or more arrays, the **distributees**, by establishing for each distributee a mapping between its index domain and the index domain of the **distribution target**, which is either a processor array or a section thereof. The distribution target is specified, after the keyword TO, in a TO-clause. The mapping between distributee and processor array can be specified either **explicitly**, as a **distribution format list**, or as an **inherited distribution**. The elements in the distribution format list are associated with the dimensions of the distributee; each element is one of the following:

1. BLOCK
2. GENERAL_BLOCK(restricted-expression)
3. CYCLIC[(specification-expression)]
4. ":"

The meaning of these elements will be discussed below. Inherited distributions will be discussed in Section 7.

Examples:

```
!HPF$ DISTRIBUTE A(BLOCK)
!HPF$ DISTRIBUTE B(CYCLIC) TO Q(1:NOP:2)
!HPF$ DISTRIBUTE C(GENERAL_BLOCK(S))
!HPF$ DISTRIBUTE (BLOCK,:) :: E,F
```

4.1 Determining an Array Distribution

Let A denote an array of rank n which is not a dummy argument, and assume that R is the associated *distribution target* (explicitly or implicitly specified). The distribution of A is specified by a list of *distribution formats*. The length of this list must be n. A *distribution format* ":" specifies that the corresponding array dimension is not being distributed. The rank of R must be n, reduced by the number of colons in the *distribution format-list*. The non-colon entries in the *distribution format list* are matched from left to right to the dimensions of R. For each such entry, a distribution function is determined according to the rules defined below. Here we assume both the array and the processor array are one-dimensional, with index domains $\mathbf{I}^A = [1 : N]$ and $\mathbf{I}^R = [1 : NP]$. We will define the functions associated with the *distribution formats* by specifying the associated distributions, which will be simply denoted by δ.

4.1.1 Block Distributions

The block distribution function is specified by the *distribution format* BLOCK; it divides the array into contiguous blocks whose sizes are identical, except possibly for the last block, which may be of a smaller size. More precisely, let $q := \lceil \frac{N}{NP} \rceil$. Then:

- $\delta(i) = \{j\}$ for all $i, 1 \leq i \leq N$, where $j = \{\lceil \frac{i}{q} \rceil\}$.
- The local index associated with element $A(i)$ in processor $R(j)$ is $i - (j-1) * q$.

4.1.2 General Block Distributions

A *distribution format* for a general block distribution is of the form GENERAL_BLOCK(G), where G is an integer array with index domain [1:M], where $M \geq NP - 1$.

A is partitioned into NP contiguous blocks. For all $i, 1 \leq i < NP$, $G(i)$ specifies the upper bound of block i. The index range associated with block 1 is $[1 : G(1)]$; for $1 < i < NP$, $[G(i-1)+1 : G(i)]$ is the index range of block i; and $[G(M-1)+1 : N]$ is the index range of block NP.

4.1.3 Cyclic Distributions

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

$$\delta(i) = \{MODULO(\lceil \tfrac{i-1}{k} \rceil, NP+1)\} \text{ for all } i, 1 \leq i \leq \text{N}$$

Cyclic distributions are specified by the *distribution format* CYCLIC. This is equivalent to CYCLIC(1).

4.2 The REDISTRIBUTE Directive

The REDISTRIBUTE directive is syntactically similar to the DISTRIBUTE directive but may appear only in the execution part of a program unit. It is used for dynamically changing the distribution of an array and may only be used for arrays that have been declared as DYNAMIC.

If an array B is redistributed, then every array A that is aligned to B is redistributed in such a way that the relationship expressed by the alignment function linking A to B is kept invariant (see Section 2.4). If B is a secondary array at the time of redistribution, then the actual alignment forest changes as follows: B is disconnected from A and made into a new degenerate tree with primary array B.

5 Alignment Directives

The ALIGN directive is used to distribute data objects indirectly, by specifying one or more direct alignment relationships and the associated alignment functions (see Sections 2.3 and 2.4).

Every axis of the alignee is specified as either ":" or "*" or an *align-dummy*, which is a scalar integer variable. If it is ":", then positions along that axis will be spread out across the matching axis of the alignment base; if it is "*", then that axis is collapsed: positions along that axis make no difference in determining the corresponding position of the alignment base. (Replacing the "*" with an align-dummy not used anywhere else in the directive would have the same effect; thus this notation is a convenience only). An align-dummy is considered to range over all valid index values for that dimension of the alignee.

Each element of the alignee is aligned with all corresponding positions of the alignment base.

5.1 Determining the Alignment Function

This section describes how an ALIGN directive specifies the alignment function associated with the direct alignment relationship between alignee and alignment base. Let

- A denote the alignee, and $\mathbf{I}^A = [L_1 : U_1, \ldots, L_n : U_n]$
- B denote the alignment base, and $\mathbf{I}^B = [L'_1 : U'_1, \ldots, L'_m : U'_m]$

The alignment function mapping $\mathbf{I}^A$ to the power set of $\mathbf{I}^B$ will be denoted by α.
Assume that the directive has the form

$$\text{ALIGN } A(s_1, \ldots, s_n) \text{ WITH } B(t_1, \ldots, t_m)$$

where

- each s_i is ":", "*", or an *align-dummy*
- each t_j is a *base-subscript*. This can be any of the following cases:
  - a *dummyless-expr*, i.e., a scalar integer expression in which no *align-dummy* occurs
  - a *dummyuse-expr*, i.e., a scalar integer expression in which exactly one *align-dummy* occurs
  - a *subscript-triplet*
  - "*"

We explain the construction of α by first applying a sequence of transformations to the directive which eliminate ":" and "*" in the alignee, and subscript-triplets as well as "*" in the *base-subscript-list.* The transformations are specified as follows:

- Assume that s_i =":" matches the subscript triplet $t_j = [LT : UT : ST]$. Then $U_i - L_i + 1 \leq MAX(INT(UT - LT + ST)/ST, 0)$ must hold. The positions in axis i of the alignee are spread out across axis j of the alignment base:

  s_i is replaced by a new align-dummy J, and t_j is replaced by the expression $(J - L_i) * ST + LT$. (This is analogous to array assignment).

- Assume that s_i ="*". Then axis i of the alignee is collapsed:

  s_i is replaced by a new align-dummy J which occurs nowhere else.

- Assume that t_j ="*". This denotes replication:

  $B(t_1, \ldots, t_{j-1}, *, t_{j+1}, \ldots, t_m)$ is replaced by the set $\{B(t_1, \ldots, t_{j-1}, k, t_{j+1}, \ldots, t_m) \mid L'_j \leq k \leq U'_j\}$.

By applying these transformations until neither the alignee nor the alignment base contain positions with either ":" or "*" we obtain:

- a *reduced alignee* of the form $A(J_1, \ldots, J_n)$, where the J_i are distinct *align-dummies.* The range of J_i is given by $[L_i : U_i]$.

- an *alignment base set ABS*, every element of which has the form $B(y_1, \ldots, y_m)$, where each y_j is either a *dummyless-expr* or a *dummy-use-expr.* The operators "+", "-", and "*" may be applied to form expressions which are linear in the *align-dummy.* Since linear expressions cannot handle some frequently occurring cases, such as truncation at either end of the alignment, we also allow the intrinsic functions MAX, MIN, LBOUND, UBOUND, and SIZE to be used in alignment functions. Each J_i may occur in at most one y_j (this excludes the possibility to specify skew alignments).

The basic rules for determining α are now as follows:

1. Select an arbitrary tuple $\mathbf{j} = (j_1, \ldots, j_n)$, where each j_i is a value in the range of J_i, and substitute j_i for each occurrence of J_i in *ABS.*

2. Evaluate all expressions in the modified set *ABS*; this evaluation is performed modulo the extent of the associated dimension of the alignment base: the value y associated with dimension j is replaced by $\hat{y} = MIN(U'_j, y)$.

Example:

```
      REAL A(1:N), D(1:N,1:M)
!HPF$ ALIGN A(:) WITH D(:,*)
```

aligns a copy of A with every column of D. The reduced alignee has the form $A(J)$, where the range of J is $[1 : N]$. For the alignment base set we obtain: ABS=$\{D(J,k) \mid 1 \leq k \leq M\}$. Hence, $\alpha(J) = \{(J,k) \mid 1 \leq k \leq M\}$ for each $J \in [1 : N]$.

Example:

```
      REAL B(1:N,1:M), E(1:N)
!HPF$ ALIGN B(:,*) WITH E(:)
```

Here, the reduced alignee has the form $B(J_1, J_2)$, where the range of J_1 is $[1 : N]$ and the range of J_2 is $[1 : M]$. For the alignment base set we obtain:ABS=$\{E(J_1)\}$. Thus, $\alpha(J_1, J_2) = \{(J_1)\}$ for each $J_1 \in [1 : N]$ and $J_2 \in [1 : M]$.

5.2 The REALIGN Directive

The REALIGN directive is syntactically similar to the ALIGN directive but may appear only in the execution-part of a program unit. It is used for dynamically changing the alignment of an array and again may only be used for arrays that have been declared as DYNAMIC.

Assume that A is the alignee, B the base array, with distribution δ_R^B, and α the alignment function determined by the REALIGN directive. Then the actual alignment forest is modified as described by the steps below:

1. If A is a primary array at the root of a non-degenerate tree immediately before execution of the REALIGN directive, then all secondary arrays associated with A are disconnected from A and made into primary arrays of degenerate trees with their current distribution.

   If A is a secondary array with associated primary array B', then A is disconnected from B'. (Note that $B' = B$ is possible).

2. A is made a new secondary array of B.

3. The distribution of A is determined as $\delta_R^A = CONSTRUCT(\alpha, \delta_R^B)$

6 Allocatable Arrays

Distribution and alignment for variables with the ALLOCATABLE attribute may be specified using DISTRIBUTE or ALIGN directives. These directives may occur in the specification-part of a program unit just as for other arrays: the associated attributes are propagated to each associated ALLOCATE statement. Such variables may also be used in REDISTRIBUTE and REALIGN directives.

In the following example, distributions are specified for the allocatable arrays A, C and D which are valid for each allocation instance. When C is allocated in the instance shown, it is given a cyclic distribution in the executable REDISTRIBUTE directive. At the time ALLOCATE is applied to an array B, the array is created according to the alignment given in the executable REALIGN statement. The actual alignment forest is modified by entering B as a new element in the position determined by the alignment relationships involving B. At the time DEALLOCATE is applied to B, the array is removed from the alignment forest and each array A directly aligned to B is made into a new tree with primary A. Note that a local array which is not declared ALLOCATABLE cannot be aligned in the specification-part of a program unit to an allocatable array.

Example:

```
      REAL,ALLOCATABLE(:,:) :: A,B
      REAL,ALLOCATABLE(:) :: C,D
!HPF$ PROCESSORS PR(32)
!HPF$ DISTRIBUTE A(CYCLIC,BLOCK)
!HPF$ DISTRIBUTE(BLOCK) :: C,D
!HPF$ DYNAMIC B,C
      ...

      READ 6,M,N
      ALLOCATE(A(N*M,N*M))
      ALLOCATE(B(N,N))
!HPF$ REALIGN B(:,:) WITH A(M::M,1::M)
      ALLOCATE(C(10000), D(10000))
!HPF$ REDISTRIBUTE C(CYCLIC) TO PR
```

7 Procedures

The distribution of dummy arguments can be specified as shown below; it can also be specified by giving an alignment to another dummy argument or a local data object in the usual way. Further, a local data object may be aligned to a dummy argument.

The alignment tree, as defined in Section 2.4, is local to a procedure. Thus, an array which is the actual argument of a procedure call is not connected with its alignment tree in the calling unit during execution of the called procedure.

If a dummy argument is redistributed or realigned during execution of the procedure, then the original distribution must be restored on procedure exit.

The distribution of a dummy argument A can be specified in four different ways:

1. **explicitly** by providing a distribution specification of the form:

   DISTRIBUTE A *d* [TO *r*]

   where *d* is a parenthesized *distribution format-list*, and *r* is the *distribution target*. Here, the distribution of the actual argument is changed, if necessary, to the distribution determined by the specification (see Section 4.1). If necessary, the distribution of A before the call has to be restored upon exit from the procedure.

2. by **inheritance**, syntactically expressed by:

   DISTRIBUTE A *

   In this case, the distribution of the actual argument is transferred into the procedure and **inherited** by A.

3. by **inheritance matching**, syntactically expressed by:

   DISTRIBUTE A * *d* [TO *r*]

   A specification of this form indicates that the distribution of the actual argument is transferred into the procedure and inherited by A. However, if this distribution does not match the above specification, then the program is not HPF-conforming.

   If this distribution attribute of the dummy is known within the calling routine (through the use of interface blocks, for example), then the language processor will arrange for remapping the actual argument to the specified distribution (and mapping it back on return from the subprogram, if necessary). If the distribution attribute of the dummy is not made available when the caller is compiled, the onus is on the programmer to arrange for proper distribution of the actual argument.

4. **implicitly**: No explicit distribution is specified (directly or indirectly). In this case, the compiler provides an implicit distribution specification.

8 The Template Directive in High Performance Fortran

In the above sections, we have presented a model for mapping of data to processor memories without using templates. We claim that the HPF template directives are limited in their applicability and give rise to serious problems in the specification of the language, without adding any significant functionality.

Template directives, which may occur only in the specification part of a (sub)program, result in the creation of a **template**. Although the language definition states that "templates are just abstract index spaces", it postulates in other places that distinct definitions of templates in the same or different scopes are to be considered as different, independent of their associated index domain. As a consequence, each template created in a program execution must be interpreted as a *tagged index domain.*

The discussion in the rest of this section does not include the so-called "natural templates" of HPF: they represent the index domain associated with an array and are thus implicitly part of our proposal. In fact, our claim could be rephrased as saying that "natural templates" are sufficient to describe all features related to distribution and alignment.

8.1 The Usefulness of Templates

Templates have been perceived to have two separate uses within the language. We discuss each of these briefly.

8.1.1 Alignment of Staggered Grids

The first use of templates is to enable the specification of alignment between arrays where there is no appropriate common index domain: this can occur whenever two or more arrays are each associated with different parts of a physical grid which do not completely overlap.

Before we discuss the general case, we consider the example posted on the HPFF Distribution mailing list by C. A. Thole:

```
      REAL U(0:N,1:N), V(1:N,0:N), P(1:N,1:N)

!HPF$ TEMPLATE T(0:2*N,0:2*N)
!HPF$ ALIGN P(I,J) WITH T(2*I-1,2*J-1)
!HPF$ ALIGN U(I,J) WITH T(2*I,2*J-1)
!HPF$ ALIGN V(I,J) WITH T(2*I-1,2*J)

      P=U(0:N-1,:)+U(1:N,:)+V(:,0:N-1)+V(:,1:N)
```

For the above code, the claim was made that:

1. *Only a template with a larger index domain than any of the arrays involved represents the nature of the physical grid structure correctly.*

2. *Therefore the template T is required to specify the relationship between the data objects precisely: in particular, it is supposed to express the fact that P(I,J) is a neighbor of U(I,J) and U(I-1,J), but not of U(I+1,J), and similarly for P and V.*

3. *The actual distribution of the template (which is deliberately omitted) is irrelevant and will be chosen in a machine-dependent manner.*

Now, note that whenever two data objects in HPF are aligned with the *same* element of a template, then the language guarantees that these objects will be mapped to the same physical processor. But in the above example, all arrays are aligned with *disjoint* elements of the template. As a consequence, only the *distribution* of the template decides the actual, physical neighborhood relation. For example, the distribution

```
!HPF$ DISTRIBUTE(CYCLIC,CYCLIC)::T
```

results in the worst possible effect, viz. different processor allocations for any two neighbors.

While an alignment relation between *arrays* in a program's data space is a relatively natural concept, the template-based code above does not establish one. Hence, this example is misleading at best, and would seem to point out a danger associated with the template concept rather than a use for it.

However, the user will certainly desire to specify a collocation of the arrays in the above code or similar codes, which can be accomplished by declaring a template of size (N+1,N+1). It is indeed not possible to correctly specify an HPF alignment (without a template) in this situation. Our extension of the HPF alignment directive (which allows restricted usage of MAX and MIN), will suffice to permit explicit alignment directives for many cases which

occur in practice, including this one. Otherwise, the distributions must be specified explicitly. Given a suitable definition of the block distribution, one way to perform the required distributions is the following:†

```
      REAL U(0:N,1:N), V(1:N,0:N), P(1:N,1:N)
!HPF$ DISTRIBUTE (BLOCK,BLOCK):: U,V,P

      P=U(0:N-1,:)+U(1:N,:)+V(:,0:N-1)+V(:,1:N)
```

The language proposal contained in this paper offers a much more general solution, by providing a generalized form of block distribution.

8.1.2 Passing Array Sections to Subroutines

The second perceived use for a template directive was to permit the explicit declaration of mappings of array sections in subroutines:

```
      REAL A(1000)
!HPF$ DISTRIBUTE A (CYCLIC(3))

      CALL SUB(A(2:996:2))
       ...

      SUBROUTINE SUB(X)
      REAL X(:)  !X inherits its distribution
```

We assume that the dummy argument in subroutine SUB inherits its distribution from the actual argument.
The question raised here is:

how can the mapping of X be declared in SUB if one wants to specify it explicitly?

Now one will, in general, *not* want to explicitly specify such a distribution: the relatively high cost associated with data movement on the current generation of parallel computers means that a subroutine will usually be written so that it is invoked with distributed arguments and the dummy arguments will indeed inherit the distribution from the actual argument as above. However, just as we write one subroutine to handle arrays of different sizes, so one expects such a subroutine to accept arrays with different distributions. In

†Here the Vienna Fortran definition of BLOCK is assumed. With the HPF definition, this will cause a problem if and only if the number of processors divides N exactly.

those cases where a subroutine is important enough to warrant a specific redistribution of its arguments, or if this should be necessary for some reason, then the language provides the constructs required to prescribe the mappings.

Templates were seen as a solution to the problem of providing distributions such as that of X above explicitly, should it be deemed necessary:

```
      SUBROUTINE SUB(X)
!HPF$ TEMPLATE T(1000)
!HPF$ ALIGN X(I) WITH T(2*I)
!HPF$ DISTRIBUTE T (CYCLIC(3))
```

The template does help to specify this distribution in the example, but at the above-mentioned cost of a loss of generality for the entire subroutine. Note, further, that the same effect can be achieved by passing the entire array A to the subroutine and either using the array section explicitly or, if it is passed as a separate argument, repeating the alignment of the argument as above:

```
      SUBROUTINE SUB(A,X)
!HPF$ REAL A(1000)
!HPF$ ALIGN X(I) WITH A(2*I)
!HPF$ DISTRIBUTE A *(CYCLIC(3))
```

(The asterisk indicates that the distribution of A is inherited). But recall that if there is another call site for this subroutine with a different actual argument for X, then neither of these solutions will be of any use. Instead, inquiry functions must be used to determine the properties of alignments and/or distributions passed into the subroutine.

The current definition of HPF further attempts to facilitate the manipulation of the distributions of sections of arrays passed to subroutines by introducing the INHERIT directive, which further removes the need for explicit use of templates in this situation (albeit at the cost of introducing a host of new syntactic and semantic difficulties).

The main reason for this problem is that the current HPF language specification has an unfortunate shortcoming: HPF **cannot** (in contrast to, for example, Kali or Vienna Fortran, which include the concept of *user-defined distribution functions*), describe explicitly every distribution that it can actually generate.

8.2 Language Problems with Templates

We now reiterate the two major problems caused by templates in the HPF language definition. Note that templates are **not** first-class objects of the language: in particular, **tem-**

plates cannot be defined as being **ALLOCATABLE**. Furthermore, they cannot be passed as arguments to subroutines.

1. Templates cannot handle allocatable arrays:

   While the shape of templates is determined at entry to a program unit and cannot be changed afterwards, an allocatable array may be subject to multiple ALLOCATE and DEALLOCATE statements, where the extents of the dimensions associated with each instance may depend on run-time and input values. There is no way in which HPF can establish a direct relationship between the shape of an instance of an allocatable array, and the shape of an associated template.

   Methods to avoid this dilemma would include the definition of allocatable templates, or of infinite templates (neither of which are a serious alternative).

2. Templates cannot be passed across procedure boundaries:

   A data object whose distribution is described by a template may be passed to a subprogram in such a way that the dummy inherits the distribution. If we need to describe the distribution of the dummy argument, then we must be able to refer to the template of the actual (see above example). In HPF this would require the passing of templates to the subprogram as well. The **INHERIT** option for dummy arguments in the current HPF definition tries to achieve exactly that, introducing an element of *maximum surprise* for the user. The above example could be written as follows:

```
      REAL A(1000)
!HPF$ DISTRIBUTE A (CYCLIC(3))

      CALL SUB(A(2:996:2))
       ...

      SUBROUTINE SUB(X)
      REAL X(:)

!HPF$ INHERIT::X
!HPF$ DISTRIBUTE X *(CYCLIC(3))
```

   The idea here is that the distribution specified for X is not the distribution of the dummy argument, i.e., the distribution of the array section A(2:996:2), but that of the array associated with the actual argument.

In contrast, the distributions defined in the language proposal of this paper (as well as in Vienna Fortran) are considered to be an attribute of an array, and they are handled that way as well. Even in the case of inherited distributions which cannot be explicitly specified, inquiry functions can be used to determine every aspect of the distribution passed into the procedure.

9 Related Work

Many of the concepts and constructs used in the above language proposal, and in the HPF specification, are not new. Processor arrays and the distribution of data to them were first used for distributed memory machines in the Kali programming language [9]. They were further refined in the Vienna Fortran language, where processor arrays could also be reshaped, now expressed by means of the HPF VIEW attribute. A major difference in the handling of processor arrays is, however, that Vienna Fortran supports the mapping of data to subsets of processor arrays and provides a canonical mapping of processor arrays to a linear processor array, to facilitate the portability of code.

The Vienna Fortran language [1, 3, 12] is based both upon Kali and upon experience gained with the SUPERB parallelization system ([7, 11, 13]); it provides the user with a wide range of facilities for mapping data structures to processors, including those proposed in this paper and user-defined distributions. Vienna Fortran was the first language in which the issues of distribution handling at subroutine boundaries were investigated in depth. It introduced the concept of inheriting and of enforcing distributions and provided an attribute to enable the user to make assertions about the distributions of actual arguments. This language was also the first to make the distinction between static and dynamic distributions.

Among other things, the mapping of data to subsets of processors and the inheritance of distributions have been implemented within the framework of the Vienna Fortran Compilation System. Two variants of the general block distribution used in this paper, but not included in HPF, have also been implemented.

The programming language Fortran D [6] proposes a Fortran language extension in which the programmer specifies the distribution of data by aligning each array to a decomposition, which corresponds to a template, and then specifying a distribution of the decomposition to a virtual machine. These are executable statements, and array distributions are dynamic only.

The Yale Extensions [4] specify the distribution of arrays in three stages: alignment, partition and a physical map. Because all these stages are modeled as bijective functions between index domains, data replication is not possible. By restricting the scope of layout

directives to phases, a block structure is imposed on Fortran 90.

Cray Research Inc. has announced a set of language extensions to Cray Fortran (cf77) [10] which enable the user to specify the distribution of data and work. They provide intrinsics for data distribution and permit redistribution at subroutine boundaries. Further, they permit the user to structure the executing processors by giving them a shape and weighting the dimensions. Several methods for distributing iterations of loops are provided.

10 Conclusions

An approach which substantially reduces the cost of developing codes for distributed memory parallel machines is to provide a set of extensions for sequential languages (in particular, Fortran and C). These extensions should be portable across a wide range of architectures and should suffice for a wide variety of algorithms. The methods by which the user may distribute data to the processors are the central feature of such a language, and should be as natural and as flexible as possible. In this paper, we have presented in detail such a model for distribution and alignment of data. This model is both simpler and more general than the current High Performance Fortran model. In particular, it does not require a template directive and has simplified the passing of distributed arguments to subroutines. On the other hand, the concept of distribution functions has been generalized. A full description of the model described in this paper can be found in [2].

References

[1] S. Benkner, B. Chapman, and H. Zima. Vienna Fortran 90. In *Proceedings of the Scalable High Performance Computing Conference 1992*, IEEE Computer Society Press, Los Alamitos, April 1992, pages 51–59.

[2] B.M. Chapman, P. Mehrotra, and H. Zima. High Performance Fortran Without Templates: An Alternative Model for Distribution and Alignment, Technical Report, University of Vienna, 1993.

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

[4] 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.

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

[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. Reprinted as Tech. Report TR90-1, Austrian Center for Parallel Computation, January 1990.

[8] High Performance Fortran Forum. *DRAFT High Performance Fortran Language Specification. Version 1.0 Draft*, January 25,1993. Technical Report, Rice University.

[9] 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.

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

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

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

[13] H. P. Zima and B. M. Chapman. Compiling for distributed-memory systems. *Proceedings of the IEEE, Special Section on Languages and Compilers for Parallel Machines (To appear 1993).* Also: Technical Report ACPC/TR 92-16, Austrian Center for Parallel Computation (November 1992).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 1993	Contractor Report

4. TITLE AND SUBTITLE
HIGH PERFORMANCE FORTRAN WITHOUT TEMPLATES: AN ALTERNATIVE MODEL FOR DISTRIBUTION AND ALIGNMENT

5. FUNDING NUMBERS
C NAS1-19480
WU 505-90-52-01

6. AUTHOR(S)
Barbara Chapman, Piyush Mehrotra, and Hans Zima

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 93-17

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-191451
ICASE Report No. 93-17

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Michael F. Card
Final Report
To appear in the Proc. of the Principles & Practices of Parallel Programming, San Diego, May 1993

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Language extensions of Fortran are being developed which permit the user to map data structures to the individual processors of distributed memory machines. These languages allow a programming style in which global data references are used. Current efforts are focussed on designing a common basis for such languages, the result of which is known as High Performance Fortran (HPF). One of the central debates in the HPF effort revolves around the concept of templates, introduced as an abstract index space to which data could be aligned. In this paper, we present a model for the mapping of data which provides the functionality of High Performance Fortran distributions without the use of templates.

14. SUBJECT TERMS
distributed memory multiprocessors; data distributions; Fortran language extensions

15. NUMBER OF PAGES
22

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std Z39-18
298-102

☆U.S. GOVERNMENT PRINTING OFFICE: 1993 - 728-064/86002