An Introduction to High Performance Fortran

JOHN MERLIN AND ANTHONY HEY

Department of Electronics and Computer Science, University of Southampton, Southampton, S017 1BJ, U.K.

ABSTRACT

High Performance Fortran (HPF) is an informal standard for extensions to Fortran 90 to assist its implementation on parallel architectures, particularly for data-parallel computation. Among other things, it includes directives for specifying data distribution across multiple memories, and concurrent execution features. This article provides a tutorial introduction to the main features of HPF.

1 INTRODUCTION

High Performance Fortran (HPF) is an informal standard for extensions to Fortran 90 to assist its implementation on parallel computers, particularly for data-parallel computations. Foremost among these extensions are directives for specifying how data are to be distributed over the "processor memories" of a multiprocessor architecture, for instance, over the local memories of a distributed memory message-passing machine, a single instruction multiple data (SIMD) machine or a workstation network, or the caches of a shared memory machine. HPF also provides extensions for expressing data parallelism and concurrency, and a number of other new features.

The language was developed between March 1992 and May 1993 by the High Performance Fortran Forum, a working group comprising representatives of most parallel computer manufacturers, several compiler vendors, and a number of government and university research groups in the field of parallel computation. The formal language definition is contained in the "High Performance Fortran Language Specification," which was published in this journal [1]. A textbook on HPF has also been published [2].

This article provides a tutorial introduction to HPF, especially to its data distribution features. On many architectures the performance of an HPF program will depend critically on its data distribution, and to a lesser extent on its use of the facilities for expressing data parallelism and concurrency. Therefore we aim to give the reader an understanding of how to use these features effectively. To this end, we attempt to give some insight into how these features may typically be implemented, and in some cases also discuss the rationale for their introduction. For reference, detailed pointers are given throughout this article to relevant text in the "High Performance Fortran Language Specification" [1], for example, [HPF p. 41 (28–48)], where the numbers in parentheses are lines numbers.

The HPF features are demonstrated using two main examples, Jacobi iteration and Gaussian elimination, as well as by a number of smaller examples. The examples use a number of Fortran 90 features that are not in Fortran 77, such as free source form, simple array syntax, and new-style declarations. Array syntax in particular helps to make the examples concise, and also has the ad-

Received November 1993
Revised November 1994

e-mail: jhn@ecs.soton.ac.uk

Any opinions and recommendations contained herein are those of the authors, and do not necessarily represent the views of the High Performance Fortran Forum.

Scientific Programming, Vol. 4, pp. 87–113 (1995)
CCC 1058-9244/95/020087-27

vantage of explicitly expressing the potential for data parallelism. However, we trust that the examples should be understandable by readers who do not have a detailed knowledge of Fortran 90.

An official subset of the language, "Subset HPF" has been defined to facilitate rapid initial implementation [HPF §8]. Some Subset HPF implementations (i.e., compilers or translators) are already available and many more are expected to appear in the near future, while it may be some time before full HPF is widely supported. Subset HPF is based on a subset of Fortran 90, which is, broadly speaking, Fortran 77 plus Fortran 90's noncharacter array features and intrinsics, dynamic memory allocation, nongeneric interface blocks, optional and keyword arguments, new-style type declarations, and various lexical and syntactic improvements [HPF §8.1]. It also includes just a subset of the HPF extensions. All of the features described in this article are in Subset HPF unless otherwise stated.

This article is organized as follows. Section 2 outlines the background and motivations for the development of HPF and gives an overview of the HPF programming model. Sections 3–5 describe various aspects of the data distribution extensions, namely basic data distribution, dummy argument distribution, and dynamic redistribution, respectively. Section 6 describes HPF's extensions for expressing data parallelism and concurrency. Section 7 summarizes the remaining HPF extensions, and Section 8 discusses some pros and cons of the HPF approach.

2 WHY HPF?

A major motivation for the development of HPF was to simplify the programming of distributed memory message-passing systems, an architectural category that includes distributed memory multiple instruction multiple data (MIMD) machines and, more recently, networks of workstations. These have proved cost-effective, scalable, versatile and capable of high performance, but have also proved very difficult to program, as we shall now describe.

2.1 SPMD Programming Model

The most popular programming model for massively parallel distributed memory architectures (i.e., those with a large number of processors) is the single program multiple data (SPMD) model. The same program, though not necessarily the same instruction stream, is executed by every processor, each operating on a part of the data.

To develop such a program, the application's data arrays must be partitioned into segments which are mapped to the processor memories, a procedure known as *distributing* the arrays. Then the computations are also distributed over the processors; typically each processor performs only those computations that define data elements that are "owned" by it, i.e., stored in its local memory. If the program running on one processor requires data that are stored in the local memory of another processor, the data must be communicated by inserting explicit send and receive statements at appropriate points in the program, which is called *message passing*. Typically, accesses to local data are much faster than nonlocal accesses (i.e., communications). Therefore, for efficiency it is important to partition the data and computations in a way that attempts to minimize communications and maximize data parallelism.

For example, consider the fragment of Fortran 90 code shown in Figure 1. This uses Jacobi's method to approximate the solution of a partial differential equation (Laplace's equation) discretized on a two-dimensional grid `a`. The boundary values of `a` are given, and the interior values are computed by an iterative procedure as follows. Starting from arbitrary initial values, in each iteration the value at every interior grid point is replaced by the average of the values of its nearest neighbors in the previous iteration, and this is repeated until none of the values change significantly from one iteration to the next. Our convergence criterion is that the change at every grid point is less than 10^{-7} of its previous value.

To adapt this to an SPMD program for a distributed memory message-passing architecture it must be modified as follows:

1. The arrays `a` and `new_a` must be distributed over the processor memories. For example, if there are 16 processors which are regarded as being logically arranged as a 4×4 processor array, `a` and `new_a` may each be partitioned into 16 blocks of size $(\lceil m/4 \rceil, \lceil n/4 \rceil)$, each of which is stored on the corresponding processor.
2. The program is modified to compute and update only the locally stored segments of the arrays.
3. Message passing is inserted to communicate data where necessary. In this example,

```
REAL  a (m,n),  new_a (m,n)

a = 0.0
CALL init (new_a)              ! set boundary elements of 'new_a'
new_a (2:m-1, 2:n-1) = 0.0     !initialise interior of 'new_a' to 0.0

DO WHILE (ANY (new_a - a > 1E-07 * a))
  a = new_a
  new_a (2:m-1, 2:n-1) = 0.25 * (a (1:m-2, 2:n-1) + a (3:m, 2:n-1)  &
                                 + a (2:m-1, 1:n-2) + a (2:m-1, 3:n))
ENDDO
```

FIGURE 1 Jacobi iteration.

since the update of each point depends on the values of its four nearest neighbors, the "edge" values of each segment must be swapped between processors in every iteration. Care must be taken about special cases; for instance, the outside edges of the overall array are not communicated. Messages must also be exchanged to evaluate the global termination condition "ANY (new_a - a > 1E-7 * a)."

Chapman et al. [3] showed a simplified version of the message passing involved in a single update step.

The need to explicitly partition data, insert message passing, handle boundary cases, etc., is a very complicated, time-consuming, and error-prone task, and it also impairs the adaptability and portability of the resulting program. Indeed, the difficulty of programming distributed memory message-passing systems has so far been a big obstacle to using them.

This situation has motivated much research in recent years towards the goal of automatic parallelisation, i.e., the automatic transformation of data parallel applications written in a standard sequential language like Fortran into SPMD message-passing programs. It has become clear that this can be at least partly achieved: If the required data distribution is specified, a compiler can automatically partition the data and computations according to this specification, and insert the necessary communications [4–6].

The really difficult part of fully automatic parallelization is to automatically determine a suitable data distribution. As we have said, an efficient data distribution must spread out the data arrays over the processors (rather than storing a copy on each processor) as much as possible in order to maximize the potential parallelism, while distributing them in such a way as to minimize communications. To determine a suitable distribution therefore requires global analysis of the program's data access patterns and their relative importance. Often this information cannot be determined statically. Much research is being conducted on this problem, but currently no satisfactory conclusion has been reached.

This situation has led to the research and development of a number of prototype parallelization systems based on language extensions for specifying data distribution such as Fortran D and Fortran 90D [7, 8], Vienna Fortran [3], Distributed Fortran 90 [9, 10], and Pandore C [11]. Ideas from these (particularly the first three) and other research projects, as well as from Fortran dialects and proposals from vendors such as Digital, Convex, Cray, IBM, Maspar, and Thinking Machines, together with inputs from a variety of other sources, have all contributed to the development of the HPF informal standard. [1] and [2] provide more detailed background and references.

2.2 HPF Features and Model

As we have indicated, the central idea of HPF is to augment a standard Fortran program with specifications describing how its data are to be distributed across multiple memories. For a MIMD multiprocessor architecture, an HPF compiler transforms this program into an SPMD code by partitioning and distributing its data as specified, allocating computation to processors according to the locality of the data references involved, and inserting any necessary data communications in an implementation-dependent manner, e.g., by message passing or by a shared memory mechanism.

An HPF program is largely single threaded, i.e., all processors execute the same code. However, a few HPF features can express "functional" parallelism, whereby different processors may execute different code, as we shall point out later in this article.

Although we have concentrated on the application of HPF to distributed memory message-passing systems, it is largely architecture independent. It can be implemented across the whole spectrum of parallel architectures: distributed and shared memory MIMD, SIMD, vector, workstation networks, etc.

Data distribution is specified by *directives*. These are structured Fortran comments that are distinguished by starting with the characters HPF$ immediately after the comment character. Being structured comments they are ignored by a standard Fortran compiler and only recognized by an HPF compiler, so an HPF program can even be compiled for a single processor machine. This is acceptable as they do not affect the semantics of a program, i.e., they do not change its computations or results (except for possibly affecting the order of computations when it is not defined by the language, for instance the order of the elemental assignments that comprise an array assignment). The data distribution directives only affect a program's performance, not its meaning.

HPF also contains a few actual syntax extensions to Fortran 90, such as a FORALL statement and construct, so an HPF program cannot be compiled by a standard Fortran 90 compiler if it uses these extensions. However, nearly all of HPF's syntax extensions will be included in the next revision of the official Fortran standard due in 1995 or 1996 [12].*

Having given a general introduction to HPF, we shall now describe it in more detail.

3 DATA MAPPING DECLARATIONS

Data mapping is the HPF term for allocating data to multiple memories. In general, this mapping may be specified in two stages:

1. Data objects may be *aligned* with other data objects or with *templates*—special virtual objects that occupy no storage, which are described and motivated later. This sets up a relation between the elements of the aligned objects, such that aligned elements are guaranteed to be mapped to the same processor(s). Thus if an array A is aligned with an array or template B, the distribution of A is determined by that of B, and only the latter is specified. In this example A is called an *alignee* and B the *align target*.
2. Templates or data objects that are not alignees are *distributed* over *abstract processors*. Distribution is the mapping of the elements of a data object or template to the memories of the abstract processors. The distribution of an align target also determines that of all the objects that are aligned with it.

A third implementation-dependent level may also be involved: associating abstract processors with real physical processors. This allows implementations the freedom to abstract the processors declared in HPF from the physical processors; for instance, the former may actually be *processes*, and an implementation may be able to execute multiple processes concurrently on each physical processor.

As we said in the last section, data mapping in HPF is specified by directives [HPF §2.3].

3.1 Alignment and Distribution

To convert the Jacobi iteration code of Figure 1 to HPF, using the data distribution described in Section 2.1, the following directives can be added to the declarations part of the program—no other changes are necessary:

```
!HPF$ PROCESSORS (4, 4)
!HPF$ ALIGN a (:,:) WITH new_a (:,:)
!HPF$ DISTRIBUTE new_a (BLOCK, BLOCK) ONTO p
```

We shall now explain these directives.

The PROCESSORS *Directive*

The PROCESSORS directive [HPF §3.7] declares and names one or more *abstract processor arrangements*, where a processor arrangement means a processor array or a scalar (i.e., single) processor. In this case a set of 16 abstract processors is declared, which are regarded as being arranged in a 4×4 array called p.

* It is likely that all HPF syntax extensions except for the EXTRINSIC attribute will be included in the next Fortran revision.

Abstract processor arrays with different shapes may be declared, in which case an HPF implementation may map them in an implementation-dependent manner onto the real physical processors. However, the only processor arrangements that are guaranteed to be supported are scalar processors and processor arrays with the same number of elements as there are physical processors. Processor arrays with the same shape are equivalent, i.e., corresponding elements refer to the same abstract processor, but otherwise there is no defined relation between different processor arrangements.

Processor arrangements are not first-class objects in HPF—they may not appear in COMMON blocks nor be passed as arguments to functions or subroutines. The only way for a PROCESSORS directive to be visible in several program units is to declare it in a module which is USEd by the program units. Otherwise, processor arrangements must be declared locally in every program unit in which they are used.

The ALIGN Directive

The ALIGN directive [HPF §3.4] relates the elements of a data array to the elements of another data array or a template (to be described later), such that elements that are aligned with each other are guaranteed to be mapped to the same abstract processor(s) regardless of the distribution directives.

The given ALIGN directive:

```
!HPF$ ALIGN  a (:,:)  WITH  new_a (:,:)
```

specifies that each element of a is aligned with the corresponding element of new_a, which means that for all subscript values i and j, element a(i,j) is mapped to the same abstract processor(s) as new_a(i,j).

An operation on two or more data elements is likely to be executed much faster if they are aligned, as it can be performed without communications by the processor that stores them locally. On the other hand, independent operations may potentially be executed in parallel if they involve data that are stored in different processor memories. Therefore, alignment should be chosen so as to try to keep elements that are accessed together in the same operation stored together (i.e., aligned) to minimize communication, while keeping elements that can be operated on independently apart to maximize data parallelism. In this simple case the same result could be achieved by distributing the two arrays alike, but in general it is not possible to achieve arbitrary linear alignments of arrays (e.g., where the elements of one array are aligned with a *subset* of the elements of another) by distribution directives alone. In any case, when alignment of arrays is intended it is clearer and safer to specify it explicitly rather than relying on it being achieved as a side effect of distribution.

In this example, a(:,:) and new_a(:,:) denote the whole of arrays a and new_a, using Fortran 90's subscript triplet notation. Unfortunately, this form of the ALIGN directive requires that the name immediately following the ALIGN keyword (the *alignee*) *must* be followed by parentheses, so the Fortran 90 shorthand for specifying a whole array by just giving its name cannot be used here.† However, it can be used in another form of the ALIGN directive that we shall introduce later.

The *alignee* (a(:,:)) must be a whole array, but in general the *align target* (new_a(:,:)) may be a regular section of an array, provided that it conforms with (i.e., has the same shape as) the alignee. This allows an array to be aligned with a regular subset of the elements of another array or template. We shall give examples of this later.

The DISTRIBUTE Directive

The DISTRIBUTE directive [HPF §3.3] specifies how a data object or template is to be distributed over an abstract processor arrangement. A data object that has been aligned (i.e., has appeared as an alignee) cannot be distributed; only an align target that is not itself aligned with anything else, or an object that has not appeared in an ALIGN directive, can be distributed. Thus, in this example only new_a can be distributed, and its distribution determines that of a, which is aligned with it.

The given DISTRIBUTE directive:

```
!HPF$ DISTRIBUTE new_a (BLOCK, BLOCK) ONTO p
```

states that each dimension of new_a is *block distributed* over the corresponding dimension of the

† Parentheses are needed after the alignee name to avoid ambiguity if blanks are insignificant, as they are in Fortran 90 fixed source form. For example, without the parentheses, "ALIGN T(:) WITH TWITHEAD(:)" could be interpreted as "ALIGN TWITHT WITH EAD"!

processor array p. In general, so-called distribution format is specified for each dimension of the *distributee* (i.e., the object that is distributed), which can be either BLOCK[(blocksize)], CYCLIC[(blocksize)] or *, where [. . .] encloses an optional item. Their meanings are as follows, where for simplicity we describe the case of a one-dimensional array distributed over a one-dimensional processor array.

1. BLOCK means that the elements are divided into blocks of consecutive elements, and the nth block is allocated to the nth processor. If the number of elements, N, is exactly divisible by the number of processors, P, then the blocks are of equal size N/P. Otherwise blocks of size $b = \lceil N/P \rceil$ are allocated to the first $\lfloor N/b \rfloor$ processors, the remaining $N \backslash P$ elements form a small block which is allocated to the next processor, and no elements are allocated to any remaining processors.

   An explicit blocksize b can be given in parentheses after the BLOCK keyword, but it must be such that the elements do not "wrap around" the processor array. To allow wrap around a CYCLIC distribution must be specified. We advise against explicitly specifying b, except in special cases, as it can give rise to errors (if $N > bP$, requiring wrap around) or inefficient processor utilization (if $N \ll bP$). If b is specified, we recommend that it should depend on N and P. directly or indirectly, to avoid these problems if N or P is changed.
2. CYCLIC means that the first element is allocated to the first processor, the second to the second processor, etc. If there are more elements than processors then the distribution "wraps around" the processor array cyclically until all the elements are allocated.

   An explicit blocksize may be specified in parentheses after the CYCLIC keyword, as for the BLOCK distribution. (By definition, CYCLIC means the same as CYCLIC(1).) In this case, however, the elements *are* allowed to wrap around the processor array, in which case the distribution is often called *block-cyclic*. CYCLIC(b) is the most general type of distribution available: BLOCK (b) is just a special case of it in which there is no wrap around. If that is the case, however, then it is more efficient to specify BLOCK(b), as the extra information that there is no wrap around considerably simplifies address calculations.

   Cyclic distributions are useful for spreading the computation load uniformly over processors in cases where computation is only performed on a subset of array elements or is otherwise irregular over an array. An example of this, Gaussian elimination, is shown later.
3. * means that the corresponding distributee dimension is *collapsed*, i.e., not distributed.

These descriptions generalize straightforwardly to multidimensional distributees and processor arrays, with the words "element" and "processor" replaced by "subscript value" and "processor subscript value."

A distribution format must be specified for every dimension of a distributee. The number of BLOCK and CYCLIC entries (with or without a blocksize) must equal the number of processor array dimensions, and the nth distributee dimension with such an entry is distributed over the nth processor array dimension. [HPF, pp. 28–29] gives some illustrated examples of distribution.

The ONTO clause may be omitted from the DISTRIBUTE directive, in which case the distribution is onto an implementation-dependent processor arrangement. Although the HPF specification says nothing on this point, it is conceivable that some implementations may allow a default processor arrangement to be specified by a command line argument or environment variable when the HPF compiler is invoked; others may have a built-in default; and yet others may require a single PROCESSORS declaration in each program unit to provide the default.

Experimenting with Data Mappings

Returning to the Jacobi iteration example, with the given data mapping directives the arrays a and new_a are distributed over processors as shown in Figure 2.

In general, to achieve optimum performance of this code fragment we should partition the arrays into blocks that are as nearly square as possible, since this maximizes the ratio of calculation to communication (as the former is proportional to the total number of points in a block, and the latter to the number of its boundary points). Therefore, depending on the array sizes (i.e., the

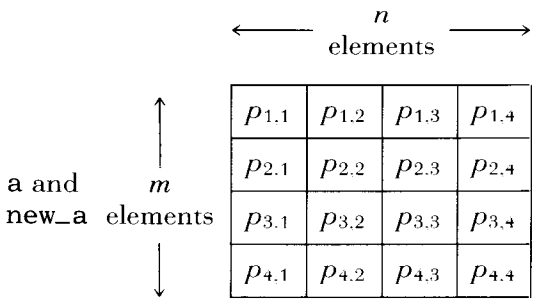


FIGURE 2 Distribution over a 2-dimensional processor arrangement.

values of m and n) and the number of processors available, other distributions may be optimal. For instance, if $m \ll n$ it might be preferable to configure the processors as a one-dimensional array and distribute only the second dimension of a and new_a (Fig. 3):

```
!HPF$ PROCESSORS p (NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE new_a (*, BLOCK) ONTO p
```

This demonstrates a benefit of the two-level mapping of data onto processors: The optimal alignment (i.e., ALIGN and TEMPLATE directives) is usually problem dependent, while the optimal distribution (i.e., DISTRIBUTE and PROCESSORS directives) often depends on the problem size and target architecture. Therefore to port an HPF program to a different architecture or optimize it for a particular problem size typically involves modifying only its distribution directives, not its alignment directives. Observe also that experimenting with different data mappings is much easier in HPF than it would be in a message-passing program!

HPF System Enquiry Functions and Specification Expressions

The last example used the function NUMBER_OF_PROCESSORS. This is a new *system enquiry* intrinsic function introduced by HPF that returns the total number of physical (as distinct from abstract) processors on which the program is executed, or, with an optional integer argument DIM, the number of processors along a specified dimension of the physical processor array [HPF §5.2, 5.6.4]. Another HPF system enquiry intrinsic function is PROCESSORS_SHAPE [HPF §5.2, 5.6.5], which returns the shape of the physical processor array. These functions return the same results throughout the duration of one program execution. They may be used in specification expressions (e.g., to declare array bounds), or indeed in any nonconstant expression. However, they cannot be used in initialization expressions (i.e., compile-time constant expressions used, for example, to initialize variables or named constants, or to declare array bounds for common block variables), as they are not necessarily compile-time constants—an HPF program may be compiled for a machine whose configuration is not known at compile time.

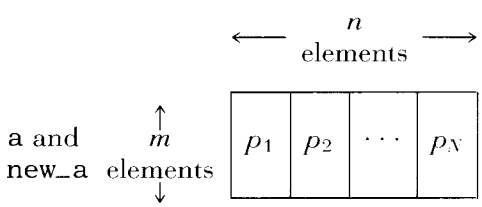


FIGURE 3 Distribution over a 1-dimensional processor arrangement.

Parameters in data mapping declarations, such as processor array sizes and blocksize in DISTRIBUTE, must be specification expressions as defined by Fortran 90, but with the extension of allowing the above HPF system enquiry intrinsic functions. In general they need not be constant. However, the mapping of common block and SAVEd variables must be constant *for the duration of a program run.* This is less stringent than requiring their mapping parameters be initialization expressions, as it allows the system enquiry functions NUMBER_OF_PROCESSORS and PROCESSORS_SHAPE to be used [HPF pp. 41 (28–48), 43(43)–44(12)].

Alternative Syntax

Finally, we mention that there is an alternative syntactic form for these mapping directives, analogous to Fortran 90's new style of declarations [HPF §3.2]. This form allows a number of attributes to be combined in the same directive, separated by a double colon (::) from the list of identifiers declared. For example, alternative forms of the directives at the start of this section are:

```
!HPF$ ALIGN WITH new_a::a
!HPF$ PROCESSORS, DIMENSION(4,4)::p
!HPF$ DISTRIBUTE (BLOCK, BLOCK) ONTO p::new_a
```

This form is more concise when several objects have to be given the same dimensions, alignment,

or distribution, as a list of names can follow the "::." Notice also that this form of the ALIGN directive allows a whole array or template to be specified by just giving its name.

3.2 Templates

In the above example we aligned a with new_a and distributed the latter, i.e., we chose new_a as the align target. Since an identity alignment is involved, we could equally well have reversed the rôles of a and new_a, and chosen a as the align target.

When several arrays have to be related by an identity alignment, rather than arbitrarily choosing one of them as the align target and aligning the others with it, or chaining them together in an arbitrary order (e.g., ALIGN a WITH b; ALIGN b WITH c; . . .), an alternative is to align them all with a *template* of the same size as the data arrays, for example:

```
!HPF$ TEMPLATE t (m, n)
!HPF$ ALIGN WITH t :: a, new_a
!HPF$ DISTRIBUTE t (BLOCK,BLOCK) ONTO p
```

A *template* in HPF is a virtual scalar or array, in other words one that occupies no storage [HPF §3.8]. Templates are declared by a TEMPLATE directive as above. Their sole function is to provide abstract objects with which data objects can be aligned and which can then be distributed, i.e., to provide intermediaries in the mapping of data objects to abstract processors. As we have seen, it is not mandatory to use templates for this purpose—data objects can be aligned directly with other data objects, and can also be distributed directly. However, there are often stylistic advantages to using templates rather than arrays as align targets, as we have just indicated.

For example, when arrays are aligned with other arrays, an arbitrarily complicated alignment tree can be constructed (see Fig. 4), which can make it difficult to identify the root object with which a given array is *ultimately aligned*. Furthermore, it can be difficult to work out the ultimate alignment of arrays that are indirectly aligned with the root, and even more difficult to establish the relative alignments of arrays on different branches of the tree. By contrast, if arrays are always aligned with templates, the alignment tree is restricted to a depth of one, and the ultimate alignment of all arrays is obvious—it is exactly as written in the ALIGN directives. This follows because templates cannot themselves be aligned: they can only be align targets.

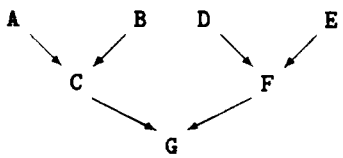


FIGURE 4 An alignment tree.

Another point is that the root object of an alignment tree indicates the maximum data parallelism that can in principle be achieved for the given program with the given alignments. This is an important characteristic of the program, so it is desirable to give the object that bears this information a separate identity, to distinguish it from the data objects. Making it a template serves that purpose.

Quite apart from these stylistic reasons for using templates, they turn out to be virtually indispensable in some situations, as we shall see later.

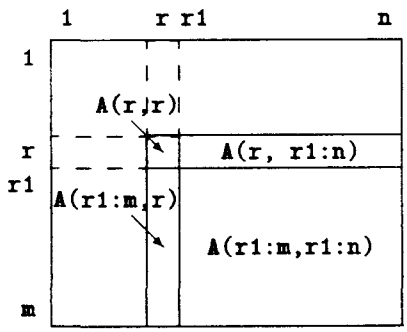
Like processor arrangements, templates are not first class objects in HPF. The restrictions on the use of processor arrangements described in Section 3.1 also apply to templates.

3.3 Gaussian Elimination

The other main example code that we shall use to illustrate HPF is the forward elimination phase of Gaussian elimination (Fig. 5). Since this code may look a little unfamiliar we briefly describe what it does. Gaussian elimination is used to solve a set of linear equations $AX = B$, where A is an $m \times m$ matrix of coefficients, and B and X are $m \times m'$ matrices composed, respectively, of a set of right-hand side vectors $\{\underline{b}_i, i = 1, m'\}$ and a corresponding set of solution vectors $\{\underline{x}_i, i = 1, m'\}$. The forward elimination stage reduces A to upper triangular form by iterating over its rows r. In interation r, row r of A is divided by $A(r, r)$, and then each row $i > r$ has $(A(i, r) \times \text{row } r)$ subtracted from it (potentially in parallel over the rows i). This sets the column below the diagonal element $A(r, r)$ to 0, so iteration over r produces the desired upper triangular form. The same operations must also be performed on B, which is done by *augmenting* matrix A with B as extra columns, giving the $(m \times n)$ matrix A that appears in the code. Another refinement is that in interation r only the sections $[r + 1{:}n]$ of the rows are operated upon, as by definition the other elements become 0 (ex-

```
INTEGER m, n, r, r1
REAL    A (m,n)
...
!-------------------------------------------------------!
! Forward Gaussian elimination of augmented matrix 'A' !
! (without pivoting). The result overwrites 'A'.       !
!-------------------------------------------------------!
DO r = 1,m
  r1 = r+1
  A(r, r1:n)     = A(r, r1;n) / A(r, r)
  A(r1:m, r1:n) = A(r1:m, r1:n) - (SPREAD (A(r1:m, r), 2, n-r)  &
                                 * SPREAD (A(r, r1:n), 1, m-r))
END DO
```

(a): Code



(b): Array sections referenced in iteration r of above code

FIGURE 5 Forward elimination phase of Gaussian elimination (without pivoting).

cept for $A(r, r)$, which becomes 1). For brevity this example omits pivoting, which would normally be used to improve numerical stability, and also omits to check that diagonal elements are nonzero.

Incidentally, notice that in order to update the whole section A(r+1:m, r+1:n) in a single data parallel operation using Fortran 90 array syntax, the SPREAD intrinsic function must be used to replicate row A(r, r+1:n) and column A(r+1:m, r) into two-dimensional arrays that conform with (i.e., have the same shape as) A(r+1:m, r+1:n). This is rather cumbersome, and HPF introduces a data parallel FORALL statement which allows it to be expressed more concisely, as we shall see later.

CYCLIC *Distribution*

Notice that the section of A that is involved in the computation diminishes as the execution progresses, i.e., iteration r only involves the section A(r:m, r:n) —see Figure 5. Therefore, if A were block distributed over the two-dimensional processor array, the area of the processor array that is utilized would diminish correspondingly. This example is therefore a candidate for cyclic distribution:

```
!HPF$ DISTRIBUTE A (CYCLIC, CYCLIC)
```

Assuming that A is larger than the processor array, this helps to spread out the workload.

Unspecified Mapping

The mapping of any data object may be left unspecified in an HPF program, in which case it will be implementation dependent. In particular, we expect that the mapping will often not be specified for scalar objects such as m, n, r, and r1 in the Gaussian elimination example. Although the default mapping for scalars is implementation dependent, on distributed memory MIMD architec-

tures it is likely that they will be *replicated*, i.e., every processor will store a copy of them. This is certainly a sensible mapping for scalars as their values are often needed by all processors, for instance, if they are used to govern control flow (as DO-loop indices, or in DO-loop control expressions, IF and DO WHILE conditions, etc.), or referenced in specification expressions, subscript expressions, etc. We shall say more about replication later.

A small refinement of this default mapping strategy is that, if the scalar is used as a DO-loop index and the implementation partitions the DO-loop iterations, allocating different iterations to different processors, the index may well be privatized for the scope of the loop. However, this will be transparent to the user, and the implementation will ensure that all copies of the scalar receive the same, correct value on termination of the DO-loop so as to preserve the program's semantics.

4 DUMMY ARGUMENT MAPPING

So far we have only considered the mapping of data objects that have their own storage, such as local and global variables. The situation is more complicated for dummy arguments, as they do not necessarily receive fresh storage, but instead serve as placeholders that are associated with a number of other objects, the actual arguments, during program execution.

HPF actually provides a number of mechanisms for specifying the mapping of a dummy argument. It can be given a *prescriptive* mapping, which forces the actual argument to acquire the specified mapping, or a *descriptive* mapping, which asserts that the actual argument is already mapped as described, or it can *inherit* its mapping from the actual argument. In fact, the mapping can be specified using any combination of these forms.

We shall demonstrate these mappings using a modified version of the Gaussian elimination code (Fig. 6), in which the update in each iteration is performed by calling a subroutine Gauss_itn. We shall now address the quation of how to specify the mapping of Gauss_itn's dummy arguments.

4.1 Prescriptive Mapping

The mapping of a dummy argument can be specified in the same way as for other data objects, using the directives already described. This constitutes a *command* to make the associated actual argument have the specified mapping, and is called *prescriptive mapping* [HPF pp. 48(18–21), 51(33–36)]. If the actual argument is not mapped as prescribed, it is automatically copied or remapped on entry to the procedure to satisfy the dummy's mapping directives, and copied or remapped back on return (unless the latter is known to be unnecessary, e.g., if the dummy argument's value is unchanged). We emphasize that the argument's mapping is always restored on return, so a data object is never permanently remapped as a side effect of passing it as an argument to a procedure [HPF p. 53(19–31)].

There are several reasons why HPF provides the prescriptive mapping facility. The most obvious is that a dummy argument may be associated

```
   INTEGER m, n, r, r1
   REAL    A (m,n)
   . . .
   DO r = 1,m
     r1 = r+1
     CALL Gauss_itn (A(r1:m, r1:n), A(r1:m, r), A(r, r1:n), A(r,r), m-r, n-r)
   END DO
   . . .
SUBROUTINE Gauss_itn (matrix, col, row, elem, n1, n2)
   INTEGER n1, n2
   REAL    matrix (n1, n2), col (n1), row (n2), elem

   row = row / elem
   matrix = matrix - SPREAD (col, 2, n2) * SPREAD (row, 1, n1)
END SUBROUTINE
```

FIGURE 6 Gaussian elimination with each iteration done by a subroutine call.

with a number of different actual arguments with different mappings, so if a particular mapping is specified for the dummy argument, some actual arguments may have to be remapped in order to satisfy it. Another reason is that in general expressions in HPF have no defined mappings, so if an actual argument is an expression it may not be possible to predict and declare its mapping. Finally, procedure boundaries are a clean and natural place for data to be remapped, as a procedure encapsulates a segment of computation for which the optimal data mapping may be different from that elsewhere.

For example, the dummy arguments of `Gauss_itn` in Figure 6 could be mapped as follows:

```
!HPF$ ALIGN col (:) WITH matrix (:, *)
!HPF$ ALIGN row (:) WITH matrix (*, :)
!HPF$ ALIGN elem    WITH matrix (*, *)
!HPF$ PROCESSORS p (4,4)
!HPF$ DISTRIBUTE matrix(BLOCK,BLOCK) ONTO p
```

Dummy argument `matrix` is associated with the actual argument `A(r1:m, r1:n)`, which is a *regular section* of an array. For the time being suppose that A is distributed (BLOCK, BLOCK) in the caller. Then in general `A(r1:m, r1:n)` occupies only a subset of the processors (i.e., the corresponding regular section of the processor array). However, specifying that dummy argument `matrix` is distributed (BLOCK, BLOCK) means that it is treated as a *whole* array which is distributed uniformly, or as uniformly as possible, over the *whole* processsor array, as described in Section 3.1. To acquire this mapping, `A(r1:m, r1:n)` will generally be copied to a temporary array with the required mapping on entry to `Gauss_itn`, and copied back on return.

In many first-generation HPF implementations, the value assigned to an array element is computed by the processor(s) that own(s) it, i.e., store(s) it in its local memory. This is called the *owner computes* rule. In that case, the distribution specified for `matrix` spreads the computation uniformly over the processor array, whereas the original distribution of the actual argument would concentrate it on the subset of processors storing `A(r1:m, r1:n)`. Therefore, the remapping reduces the computation time, as the work is distributed over more processors so each has less to do. We say that the processors are well *load balanced*. However, to this computation time must be added the time for the data remapping at the beginning and end of `Gauss_itn`, so it is uncertain whether the remapping reduces the overall execution time—that can only be determined by measurement or estimation. It would certainly not be reduced if A were distributed (CYCLIC, CYCLIC) in the caller, as we suggested in Section 3.3, since then the section `A(r1:m, r1:n)` would already be distributed as uniformly as possible over the processors so the overhead of data remapping would not be offset by an improved load balance. In that case it would be better for `matrix` to inherit its mapping from the actual argument so that no data movement occurs. Sections 4.2 and 4.3 will explain how to specify that.

Replication

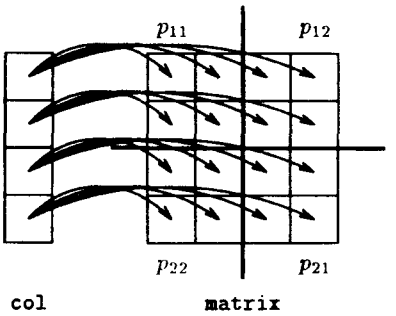
`col`, `row`, and `elem` are aligned with `matrix`. The `'*'`s that appear as dimension entries in their `ALIGN` directives mean that the alignment is independent of the subscript values in those dimensions. Incidentially, this feature of the `ALIGN` directive is not specific to dummy arguments—it can be used in any context.

For example,

```
!HPF$ ALIGN col (:) WITH matrix (:, *)
```

means that `col(i)` is aligned with `matrix(i,j)` for all subscipt values `i` and `j`, i.e., each element of `col` is aligned with a whole row of `matrix`. This in turn means that when `matrix` is distributed, `col` is copied, or *replicated*, over the processor array dimension that the second dimension of `matrix` is distributed over [HPF pp. 34(42)–35(40)]. With the given `DISTRIBUTE` directive, `col` is distributed over the first dimension of processor array `p` in the same manner as the first dimension of `matrix`, but is replicated over the second dimension of `p`, so each column of the processor array stores a complete copy of `col`. This is shown in Figure 7, using a 2×2 processor array for simplicity.‡ Similarly, the other `ALIGN` directives mean that `row` is replicated over the first dimension of `p` (so each row of `p` has its own copy of `row`) and `elem` is replicated over both dimensions of `p` (so every processor has its own copy of `elem`).

‡ Incidentally, the `ALIGN` directive should not be taken too literally in the case of replication. For example, if `matrix` has size (4,4), the `ALIGN` directive in Figure 7 suggests that each processor stores *two* identical copies of part of `col` (e.g., `(p(1,1)` stores two copies of `col(1:2)`, etc.—see Figure 7). There is an obvious optimization!




```
!HPF$ ALIGN col (:) WITH matrix (:,*)
!HPF$ PROCESSORS p (2,2)
!HPF$ DISTRIBUTE matrix (BLOCK, BLOCK) ONTO p
```

FIGURE 7 Replicating `col` over the second dimension of `matrix` (and thus processor array `p`).

Replicating `row`, `col`, and `elem` in this way means that the body of `Gauss_itn` can be executed without any communications. The array assignments in `Gauss_itn` are equivalent to the following elemental assignments performed for all values of subscripts `i` and `j`:

```
row (i) = row (i) / elem
matrix(i,j) =matrix(i,j) -col (i) *row(j)
```

The given alignments ensure that for every element assigned, the variables referenced in the right-hand side expression are stored on the same processor. Indeed, the SPREAD intrinsic functions in the original array assignments are a strong hint that replication is called for.

Replicating a variable has the advantage that its value can be read by multiple processors without communication, but may complicate its updating, as all copies must be updated. This is necessary because all copies must be kept *consistent*, i.e., they must all have the same value at any point in the program, because semantically there is just one copy of any given variable in the HPF program. For example, consider the first assignment above, to vector `row`. Each row of the processor array stores a copy of it, and all copies must be updated. If the HPF implementation uses the owner computes rule, then every processor computes the right-hand side expressions for all elements of `row` that it owns, so identical computations are performed by every row of the processor array. In this particular case that is not a drawback, however, as the execution time is the same as it would be if only one row of processors stored and updated `row` (e.g., if "`ALIGN row(:) WITH matrix(1,:)`" were specified), since all rows of the processor array do this operation in parallel.

As was the case for dummy argument `matrix`, the actual arguments associated with `row`, `col`, and `elem` do not have the prescribed mapping and so must be copied into and out of `Gauss_itn` (though an optimizing implementation might not copy back `col` and `elem` as they are not updated).

Incidentally, one might intuitively expect to be able to replicate an object by means of a `DISTRIBUTE` directive alone, but HPF syntax does not allow for that. To replicate an object it must first be aligned with a higher-dimensional object, as above. If there is no suitable data object to serve as the align target, a template can be declared for this purpose.

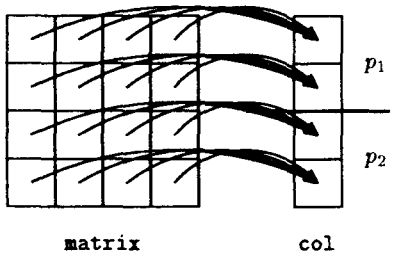
Collapsing

While we are on the subject, we shall briefly digress to mention a few remaining aspects of the `ALIGN` directive. First, "`*`" can appear as a dimension entry in an alignee, as well as an align target, with the same meaning—the alignment is independent of subscript values in that dimension. For example, consider:

```
!HPF$ ALIGN matrix (:, *) WITH col (:)
!HPF$ ALIGN row (*) WITH col (*)
!HPF$ ALIGN elem    WITH col (*)
!HPF$ PROCESSORS p (16)
!HPF$ DISTRIBUTE col (BLOCK) ONTO p
```

The first directive means that `matrix(i,j)` is aligned with `col(i)` for all subscript values `i` and `j`, i.e., a whole row of `matrix` is aligned with each element of `col`. Therefore, every element in a given row of `matrix` will be mapped to the same processor(s) (since the row is aligned with a single element of `col`, which cannot be split across multiple processors whatever `col`'s distribution: see Fig. 8). This is called *collapsing* the rows of `matrix` [HPF p. 32(28–33)]. It means that operations and assignments involving different elements in the same row will be performed without communications, at the expense of preventing concurrent operations and assignments on the elements of a row.

Collapsing and replication may be combined as in the second directive above, which means that `row(i)` is aligned with `col(j)` for all `i` and `j`, i.e., every element of `row` is aligned with each element of `col`. In other words, `row` is collapsed and




```
!HPF$ ALIGN matrix (:,*) WITH col (:)
!HPF$ PROCESSORS p (2)
!HPF$ DISTRIBUTE col (BLOCK) ONTO p
```

FIGURE 8 Collapsing rows of `matrix` onto `col` (and thus processor array `p`).

then replicated over `col`. Therefore, every processor over which `col` is distributed will store a complete copy of `row`.

`col` is distributed over a one-dimensional processor array, so the net effect of the above directives is to distribute the dummy arguments over a one-dimensional processor array in such a way that `Gauss_itn` can execute without communications.

Unlike replication, collapsing *can* be expressed directly by the DISTRIBUTE directive as well as via alignment. For example:

```
!HPF$ ALIGN matrix (:, *) WITH col (:)
!HPF$ DISTRIBUTE col (BLOCK)
```

is equivalent to:

```
!HPF$ DISTRIBUTE matrix (BLOCK, *)
!HPF$ DISTRIBUTE col (BLOCK)
```

In fact, the whole set of directives above is equivalent to the first set given in this section, with the modifications of distributing `matrix` (BLOCK, *) rather than (BLOCK, BLOCK) and changing the PROCESSORS declaration.

ALIGN *Dummy Variables*

Finally, an alternative to using ":" and "*" as dimension entries in alignees is to use dummy variables (e.g., `i` and `j`), as we did informally above to explain the meaning of "*" [HPF pp. 34(1)–36(3)]. Different dummy variables must be used in different alignee dimensions. If a dummy variable appears in the alignee but not the align target it is equivalent to a "*" in that alignee dimension. (However, it is not possible to replace a "*" in an *align target* by a dummy variable: [HPF p. 35(15–33)] explains why.) If a dummy variable d replaces a ":" in the alignee, then one dimension of the align target must contain an expression $f(d)$ that is linear in d. This means that subscript value s in the relevant dimension of the alignee is aligned with subscript value $f(s)$ in the corresponding dimension of the align target. Since $f(s)$ is linear in s, it generates a regular section when applied to the complete range of subscript s, so this form is equivalent to the subscript triplet form that we have used so far. For example, with the declarations REAL `a(8), b(8), c(64)`:

```
!HPF$ ALIGN a (i) WITH b (i)
! means ALIGN a(:) WITH b(:)
!HPF$ ALIGN b (i) WITH c (3*i + 21)
! means ALIGN b(:) WITH c(24:45:3)
```

since `i` takes subscript values in the range [1:8] for `a` and `b`.

Dummy variables need not appear in the same order in the alignee and align target, so it is possible to *permute* dimensions in the alignment mapping, e.g.:

```
!HPF$ ALIGN c (i, j) WITH d(j, i)
```

This cannot be specified using subscript triplets alone, as ":"s in the alignee are matched with subscript triplets in the align target in order of appearance. This is the main reason for using dummy variables—otherwise the ":" and "*" notation is often clearer and more succinct. Both forms can be mixed in the same directive, so the use of dummy variables can be restricted to just those dimensions where it is required for dimensional permutation.

Having digressed to discuss some features of alignment, we shall now return to the main subject of this section—dummy argument mapping.

4.2 Descriptive Mapping

An asterisk may precede certain clauses in mapping directives for dummy arguments, namely the align target in an ALIGN directive, e.g.:

```
!HPF$ ALIGN d (:) WITH *t (:)
```

and the distribution format list and/or processors name in a DISTRIBUTE directive, e.g.:

```
!HPF$ DISTRIBUTE d *(BLOCK) ONTO *p
```

where d must be a dummy argument name in both examples. Such clauses are called *descriptive*, and constitute an *assertion* that any actual argument associated with that particular dummy argument already has the described mapping characteristics. No run-time checking or remapping is performed within the procedure to satisfy these descriptive clauses. If the actual does *not* have the described mapping, the program is erroneous and its behavior is undefined [HPF pp. 48(23–32), 50(13)–51(28), 52(1–24)].

For example, suppose that in the Gaussian elimination example of Figure 6, array A is distributed as follows:

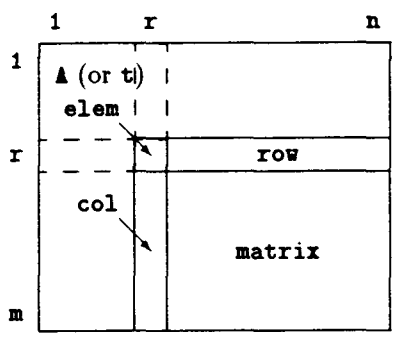
```
!HPF$ PROCESSORS p (10,10)
!HPF$ DISTRIBUTE A(CYCLIC,CYCLIC) ONTO p
```

Sections of A are passed as actual arguments to subroutine Gauss_itn, where they are associated with dummy arguments matrix, col, row, and elem. The mapping of these dummy arguments can be described with the help of a template, as shown in Figure 9, so that it corresponds exactly with that of the array section actual arguments. To help describe the mapping we have modified Gauss_itn's argument list slightly, passing in three arguments m, n, and r rather than the two array size arguments n1 and n2 used before.

This example features more general cases of alignment than we have encountered before, in which the alignee is aligned with a *regular section* of the align target, specified using the normal Fortran 90 subscript triplet notation. Incidentally, one cannot specify a regular section of the *alignee*—only of the align target. Ignoring dimensions containing "*" entries or align dummy variables, the alignee must conform with the specified regular section of the align target (i.e., corresponding dimensions must have the same number of elements), and each element of the alignee is aligned with the corresponding element of the target. As normal for Fortran 90 regular sections, any dimension of the align target can contain a scalar subscript rather than a subscript triplet, which allows a data object to be *embedded* in a higher dimensional array or template (e.g., "ALIGN col(:) WITH t(r+1:m, r)" in Fig. 9). Also as normal for regular sections, a subscript triplet can specify a stride, and this may even be negative so

```
      SUBROUTINE Gauss_itn (matrix, col, row, elem, m, n, r)
        INTEGER m, n, r
        REAL    matrix (m-r, n-r), col (m-r), row (n-r), elem
!HPF$   TEMPLATE t (m, n)
!HPF$   ALIGN matrix (:,:) WITH * t (r+1:m, r+1:n)
!HPF$   ALIGN col (:)      WITH * t (r+1:m, r)
!HPF$   ALIGN row (:)      WITH * t (r, r+1:n)
!HPF$   ALIGN elem         WITH * t (r, r)
!HPF$   PROCESSORS p (10,10)
!HPF$   DISTRIBUTE t (CYCLIC, CYCLIC) ONTO p
```

(a): Code



(b): **Layout of dummy arguments with respect to array A (or template t)**

FIGURE 9 Descriptive mapping of Gauss_itn's dummy argument.

as to reverse the sense of the mapping (e.g., "ALIGN a(:) WITH b(10:2:-2)"). An identity alignment (e.g., "ALIGN a(:,:) WITH new_a (:,:)") is just a special case in which the regular section selected from the align target is the whole of the target array. At the opposite extreme, a scalar can be aligned with a single element of an array or template (e.g., "ALIGN elem WITH t(r,r)" in Fig. 9).

Notice that the use of a template is almost indispensable in this example. The dummy arguments are associated with regular sections of an array A, so their mapping can only be described by aligning them with equivalent regular sections of an array with the same dimensions as A. (In general they cannot be described by DISTRIBUTE directives alone, for instance). However, there is no such data array within Gauss_itn to serve as the align target. One possible solution would be to declare such an array within Gauss_itn specially for this purpose, but that would waste storage, obscure the code, and perhaps cause the compiler to warn that a variable is declared but not used! Another possibility would be to pass the whole of array A itself into Gauss_itn as another argument, but that would make it pointless to also pass sections of it. Therefore, a template can be declared to serve this purpose, avoiding all of these drawbacks: It occupies no storage, and has no actual existence as a real data object in the program.

Mixing Descriptive and Prescriptive Directives

It is possible for some dummy arguments to have descriptive mapping while others have prescriptive mapping. For example:

```
!HPF$ TEMPLATE t (m, n)
!HPF$ ALIGN matrix(:,:)WITH*t(r+1:m,r+1:n)
!HPF$ ALIGN col (:)     WITH matrix (:, *)
!HPF$ ALIGN row (:)     WITH matrix (*, :)
!HPF$ ALIGN elem        WITH matrix (*, *)
```

describes the mapping of the actual argument associated with dummy argument matrix, but *prescribes* that dummy arguments col, row, and elem are replicated over matrix as they were in Section 4.1 (which implies that the corresponding actual arguments have to be copied in and out, but avoids communications within the body of Gauss_itn).

Pros and Cons of Descriptive Mapping

The only reason to use descriptive rather than prescriptive directives is for optimization purposes. The above examples could equally well have used *prescriptive* directives, by omitting the "*"s in front of the various clauses; remapping would still not occur as the actual arguments already have the prescribed mapping. The only difference is that without the "*"s the dummy argument mapping would typically be checked on entry to determine whether remapping is necessary, while this check is omitted in the descriptive case, obtaining a small time saving. However, descriptive directives must be used with care; since they *assert* the mapping, they can introduce errors into an otherwise correct program.

Having said this, we shall see in Section 4.4 that descriptive directives can be used safely if the procedure in which they appear has an explicit interface wherever it is called.

4.3 Inherited and Transcriptive Mapping

Suppose that Gauss_itn is a library routine and that we want it to accept any mapping for its arguments and not to remap them. In other words, we want the dummy arguments to inherit their mapping from the corresponding actual arguments.

This can be specified by the INHERIT directive [HPF §3.9]:

```
!HPF$ INHERIT matrix, col, row, elem
```

If no other information is provided about these dummy arguments, the associated actual arguments can have *any* mapping and will not be remapped—even if they are array elements or sections. In general, the compilation system will generate code to handle any mapping for the arguments (unless it can determine the possible actual argument mappings).

Some dummy arguments may have inherited mapping while others have prescriptive or descriptive mapping. Other data objects, including other dummy arguments, can be aligned with dummy arguments with inherited mapping. For example:

```
!HPF$ INHERIT matrix
!HPF$ ALIGN col (:) WITH matrix (:, *)
!HPF$ ALIGN row (:) WITH matrix (*, :)
!HPF$ ALIGN elem    WITH matrix (*, *)
```

inherits the mapping of matrix and replicates col, row, and elem over dimensions of it.

The basic idea of inherited mapping is straightforward and very useful. However, this concept is considerably complicated by the possibilities allowed in HPF of inheriting some characteristics of a dummy argument's mapping and prescribing or describing others, which we now describe. Unfortunately Subset HPF only includes the case where INHERITed dummy arguments have their distribution explicitly prescribed or described, which we shall consider next.

Specifying Both INHERIT *and* DISTRIBUTE

A dummy for which INHERIT is specified may optionally also appear in a DISTRIBUTE directive, thought it may not be aligned (i.e., appear as an alignee). In that case, the DISTRIBUTE directive refers to the distribution of the *template* to which the actual argument is *ultimately aligned*, rather than the distribution of the actual argument itself.§ The *alignment* of the actual argument to its ultimate template is not changed, even if the actual argument is an array element or regular section; in fact, that is the essential meaning of the INHERIT directive [HPF pp. 45(37)–47(41)].

For example, dummy argument matrix is associated with an actual argument that is a regular section of an array, namely A(r1:m,r1:n). Since A is not explicitly aligned (as it is distributed directly—see the second paragraph of Section 4.2), it is considered to be ultimately aligned with itself, and thus its ultimate template is also the array A itself [HPF p. 22(42–43)]. Therefore if "INHERIT matrix" is specified, then any DISTRIBUTE directive for matrix actually refers to the array A. Thus:

```
!HPF$ DISTRIBUTE matrix *(CYCLIC, CYCLIC)
```

asserts that A has a cyclic distribution, while

```
!HPF$ DISTRIBUTE matrix (BLOCK, BLOCK)
```

sets up a template with the same dimensions as A but with (BLOCK, BLOCK) distribution, and aligns a copy of A(r1:m,r1:n) with it in the same way that A(r1:m,r1:n) is aligned with A. Therefore, this combination of directives allows the alignment of an argument to its ultimate template to be preserved, but the distribution of that template to be asserted or changed.‖

Actually, it turns out that DISTRIBUTE can only be used in conjunction with INHERIT when the dummy has the same rank as the actual argument's ultimate template. For example, the scalar dummy argument elem is associated with the actual argument A(r,r) whose ultimate template is the two-dimensional array A. However, it is illegal to specify "DISTRIBUTE elem (BLOCK, BLOCK)," as elem itself is scalar [HPF p. 26, 4th constraint]. The same applies to the one-dimensional arguments col and row. This limits the usefulness of this combination of directives!

Transcriptive Distribution

The reverse combination, inheriting distribution characteristics but not necessarily alignment, is catered for by using asterisks in the DISTRIBUTE directive in place of the distribution format and/or processors name. For example:

```
!HPF$ DISTRIBUTE matrix * ONTO *
```

means that matrix's distribution format and the processor arrangement over which it is distributed are inherited from the actual argument. (However if INHERIT is not specified, and the actual is a regular section or is otherwise embedded into a template, its alignment will change so that it is spread out over the processor array, as though a new array were declared.) Clauses in a DISTRIBUTE directive consisting of just asterisks are called *transcriptive* [HPF pp. 48(33)–49(42)]. They are *not* included in Subset HPF.

Transcriptive and other forms can be mixed. For example:

```
!HPF$ PROCESSORS p (5,20)
!HPF$ DISTRIBUTE matrix * ONTO p
```

means that matrix inherits its distribution format but is distributed over a prescribed processor ar-

§ Ultimate alignment is explained in Section 3.2 and in [HPF p. 22, last paragraph].

‖ If changed, the distribution is restored on return from the procedure (see the beginning of Section 4.1.).

rangement p, i.e., the actual may have been distributed over a different processor arrangement, in which case it will be redistributed over p using the same distribution format as before.

```
!HPF$ PROCESSORS p (10,10)
!HPF$ DISTRIBUTE matrix * ONTO *p
```

asserts that the actual is distributed over processor arrangement p, but its distribution format is inherited and could be anything.

```
!HPF$ DISTRIBUTE matrix(BLOCK,BLOCK) ONTO *
```

means that `matrix` is to be prescriptively block distributed onto whatever processor arrangement the actual was distributed onto.

These three forms of dummy argument mapping, prescriptive, descriptive, and transcriptive or inherited, can be mixed freely, except that a dummy argument appearing as an alignee in an ALIGN directive cannot also appear in an INHERIT or DISTRIBUTE directive.

4.4 Explicit Interfaces

Finally, we consider argument mapping from the viewpoint of the caller of a procedure. Fortran 90 introduces to Fortran the possibility of making the *interface* of a procedure explicit in the caller, i.e., of providing the caller with complete information about the procedure's dummy arguments and, for a function, its result (such as their types, shapes, whether they are used as input and/or output arguments, etc.). The interface is automatically explicit for internal and module procedures, and can be made explicit for external procedures by declaring an "INFERFACE block" that contains the required information.

In HPF, if an explicit interface includes the mapping directives for the dummy arguments, then the caller will automatically remap or copy the actual arguments (for the duration of the procedure call) as necessary to satisfy them. This applies even if the mapping directives are *descriptive*; the caller treats them as prescriptive and performs any remapping necessary to satisfy them. Therefore, within the procedure the descriptive directives are guaranteed to be satisfied and so cannot be in error [HPF p. 45 (13–27)].

This suggests a "clean" optimization that programmers can apply to their HPF programs: ensure that all procedure calls have an explicit interface that includes the dummy argument mapping directives, and change all *pres*criptive directives for dummy arguments to *des*criptive ones.

5 DYNAMIC REMAPPING

The directives that we have considered so far are declarations. There are also executable forms of the ALIGN and DISTRIBUTE directives, namely REALIGN and REDISTRIBUTE, which dynamically remap data during program execution. They can only appear among the executable statements. Only the standard, prescriptive, forms of the directives are allowed; for dummy arguments, the descriptive and transcriptive forms cannot be used (as they would not make sense in the context of remapping). If a dummy argument is dynamically remapped, its original mapping is automatically restored before the procedure returns, so an actual argument cannot be remapped as a side effect of a procedure call [HPF p. 53 (19–31)]. This does not apply to variables declared in modules, however; if they are dynamically remapped within a procedure, their new mapping is preserved when the procedure returns. Common block and SAVEd variables cannot be dynamically remapped [HPF p. 37 (20–25)].

An object that has been aligned cannot be REDISTRIBUTEd, just as it cannot appear in a DISTRIBUTE directive. Therefore an object appearing in an ALIGN directive can only be redistributed if it is at the root of its particular alignment tree [HPF p. 22 (46–48)]. When such an object is redistributed it "carries" with it all the arrays aligned to it, so the alignment relations are preserved [HPF p. 25 (44–48)]. Therefore, REDISTRIBUTE may potentially result in a lot of data movement!

Conversely, an object cannot be REALIGNed if it is the root of an alignment tree (i.e., if anything else is *ultimately* aligned with it). Alignees, including interior nodes of an alignment tree, and objects not explicitly aligned *can* be realigned [HPF p. 22 (46–48)]. Realignment of a data object only affects that object—if it is an interior node of an alignment tree, it does not "carry" the objects that were aligned with it, as they are regarded as being actually aligned with the root of the alignment tree rather than with the object in question, the latter serving only as an intermediary in the description of the alignment [HPF p. 22 (37–41)].

Any object that may be subject to REALIGN or REDISTRIBUTE directives must be specified in a DYNAMIC directive [HPF §3.5], e.g.:

```
!HPF$ DYNAMIC A, B
```

As a toy example of dynamic remapping, consider the following. We have already indicated that block distribution is optimal for Jacobi iteration, while cyclic distribution is better for Gaussian elimination. Suppose that for some reason we wished to perform a phase of Jacobi iteration followed by a phase of Gaussian elimination on the same array. Then it may well be worthwhile to redistribute the array between these phases so that it is optimally mapped for each:

```
      REAL a (m,n)
!HPF$ DYNAMIC, DISTRIBUTE(BLOCK,BLOCK)::a

      ... Jacobi iteration phase
!HPF$ REDISTRIBUTE a (CYCLIC, CYCLIC)
      ... Gaussian elimination phase
```

Because dynamic remapping is potentially such an expensive operation it is only likely to be worthwhile between fairly large phases of computation, as in the above example. It is unlikely to be worthwhile just for the sake of a *single* array assignment. In general it is better to rely on the HPF implementation to generate the necessary communications for that, rather than to explicitly remap the data in order to minimize those communications, since explicit remapping moves the whole of an array while only a relatively small amount of data may need to be communicated to implement the assignment. However, as always, experiment is the best judge of the optimal mapping and remapping strategy.

The REALIGN, REDISTRIBUTE, and DYNAMIC directives are not in Subset HPF.

Allocatable Arrays and Pointers

Variables with the ALLOCATABLE or POINTER attribute may appear in ALIGN and DISTRIBUTE directives, in which case the mapping directives take effect when storage is allocated for the variables in an ALLOCATE statement. If the default mapping provided by such directives is inappropriate, an ALLOCATE statement may be immediately followed by a REALIGN or REDISTRIBUTE directive which will override the declared default.¶ In that case the variable must have the HPF DYNAMIC attribute.

Figure 10 shows an example of the use of allocatable arrays. Recall that in Section 4.1 we redistributed sections of an array by passing them as arguments to a procedure whose corresponding dummy arguments had prescriptive mapping directives. This can be achieved without a procedure call by allocating arrays with the appropriate size and distribution and assigning the array sections to them. (We must assign the sections to whole arrays because one cannot directly remap *sections* of an array, and the destination arrays must be allocatable because they have different sizes in different iterations.) This example is not very elegant, however, since copying the array sections to and from new variables serves no purpose except for its side effect of remapping them. It is less obtrusive to achieve the remapping via the procedure interface as in Section 4.1.

HPF §3.6 gives more details about mapping pointers and allocatable arrays in HPF. Incidentally, pointers are not in the Fortran 90 subset included in Subset HPF.

6 CONCURRENT EXECUTION FEATURES

Fortran 90 already contains a rich set of features for expressing data parallelism, namely its array syntax and elemental and array intrinsic functions. Since data parallelism and concurrent execution are central to HPF, it introduces a number of extra facilities for expressing them, namely a FORALL statement and construct, PURE procedures, and an INDEPENDENT directive. We shall describe them in this section.

6.1 FORALL Statement and Construct

The FORALL statement [HPF §4.1] allows a data parallel assignment to a group of array elements to be expressed in terms of its constituent elemental assignments. For example:

```
FORALL (i=1:10) A(i) = B(i) + C(i+2)
```

¶ Admittedly this prescription is somewhat inelegant. It was devised because of a desire to restrict the ALIGN and DISTRIBUTE directives to the declarations part of a program unit, thus preventing their use in conjunction with an ALLOCATE statement.

```
      INTEGER m, n, r, r1
      REAL    A (m,n),  elem
      REAL, ALLOCATABLE :: matrix (:,:), row (:), col (:)
!HPF$ ALIGN  col (:)  WITH matrix (:, *)
!HPF$ ALIGN  row (:)  WITH matrix (*, :)
!HPF$ DISTRIBUTE matrix (BLOCK, BLOCK)
      ...
      DO r = 1,m
        r1 = r+1
        ALLOCATE (matrix (m-r, n-r), col (m-r), row (n-r))
        matrix = A (r1:m, r1:n)
        col    = A (r1:m, r)
        row    = A (r, r1:n)
        elem   = A (r,r)

        row = row / elem
        matrix = matrix - SPREAD (col, 2, n-r) * SPREAD (row, 1, m-r)

        A (r1:m, r1:n) = matrix
        A (r, r1:n)    = row
        DEALLOCATE (matrix, col, row)
      END DO
```

FIGURE 10 Using allocatable arrays to remap array sections.

has the same meaning as the array assignment `A(1:10) = B(1:10) + C(3:12)`.

It is helpful to introduce some terminology for the parts of a FORALL statement. In the last example, `i` is called the *FORALL index*, the part in parentheses which declares the FORALL index and its range of values is called the *FORALL header*, and assignment statement governed by the FORALL header is called the *FORALL assignment*.

A FORALL looks somewhat like a DO-loop over array element assignments (or at least, a FORALL construct looks like that!). However, it has the same semantics as an array assignment: The expression on the right-hand side of the FORALL assignment is evaluated in parallel for *all* FORALL index values, and *then* the results are assigned in parallel to the corresponding variables, so the right-hand side expression always uses old values of array elements. Thus:

```
FORALL (i=2:9) &
  A(i) = 0.5 * (A(i-1) + A(i+1))
```

sets each of the elements `A(2)-A(9)` equal to the average of the *old* values of its nearest neighbors. It is equivalent to the array assignment:

```
A(2:9) = 0.5 * (A(1:8) + A(3:10))
```

but not to the apparently similar DO-loop:

```
DO i=2,9
  A(i) = 0.5 * (A(i-1) + A(i+1))
ENDDO
```

Incidentally, it is misleading to use the term "iterations" for the executions of the individual FORALL assignments, as that term implies sequential rather than parallel execution. In this article we use the term *instance* for this purpose, namely to mean an execution of a FORALL assignment or the body of a FORALL construct for a particular combination of FORALL index values, but it is not in standard usage—currently there does not appear to be a generally accepted term for this purpose.

The FORALL header can declare multiple indices. The general form for specifying the range of values of a FORALL index is $l:u[:s]$, where l, u, and s are scalar integer expressions for the lower bound, upper bound, and stride, respectively, and [...] denotes an optional item. l, u, and s must not depend on FORALL indices, so the "index space" is rectangular (although this can select nonrectangular array sections as we shall see).

A FORALL assignment need not be scalar—it can be an array assignment. Furthermore, sub-

scripts in a FORALL assignment can be general expressions—they are not constrained in any way. The only condition is that a FORALL statement must not assign multiple values to any element. This condition is imposed because FORALL statements and constructs are intended to be deterministic, as are Fortran 90 array assignments, meaning that the value assigned to each element of the assignment variable is well defined even though the order of the elemental assignments is undefined. The corresponding condition for array assignments in Fortran 90 is that, if an irregular section is assigned, all of its elements must be distinct. Thus:

```
FORALL (i=1:10) A(index(i)) = B(i)
```

is legal only if `indx` contains no repeated values.

```
FORALL (i=1:10, j=1:5) A(10*i+j) = C(i)
```

is legal (assuming that the generated subscripts are in range) as there are no duplicated elements on the left-hand side. It should be apparent that quite general sets of elements can be assigned by a FORALL statement!

The FORALL header can also contain a scalar logical expression called a *mask* expression, in which case the FORALL assignment, including the evaluation of its right-hand side, is only executed for those combinations of index values for which the mask expression evaluates to .TRUE.. This gives the FORALL statement a similar functionality to the WHERE statement. Thus:

```
FORALL (i=1:10, A(i) > 0.0) &
  A(i) = 1.0 / A(i)
```

is equivalent to:

```
WHERE(A(1:10)>0.0)A(1:10)=1.0/A(1:10)
```

It may seem that FORALL duplicates the functionality already provided by Fortran 90's array syntax. However, the FORALL statement is often clearer and more concise, and actually provides greater functionality, allowing more general array regions, access patterns, and expressions to be described. Therefore, it allows the explicit expression of data parallel assignment in more general cases than array syntax can handle. Without it, the programmer would be forced to use sequential syntax (such as elemental assignments in DO-loops) in these cases, which hides the data parallelism and requires that a compiler perform extensive analysis to reveal it. This reduces the chances of concurrent execution, as it is often impossible for a compiler to determine statically whether DO-loop iterations can be performed concurrently.

The following are some examples of situations where FORALL is either more convenient than array syntax, or indispensable, for expressing data parallel assignments:

1. When dimensional permutation is involved. For example:

```
FORALL (i=1:n, j=1:n, k=1:n) &
  A(i,j,k) = B(k,j,i)
```

   is clearer than the Fortran 90 equivalent, which requires the RESHAPE intrinsic:

```
A = RESHAPE (B, ORDER = (/3,2,1/))
```

2. To avoid the conformance rules for array assignments. For example, the following array assignment from the Gaussian elimination code of Figure 5 requires the use of the SPREAD intrinsic function so that all array sections conform:

```
A(r1:m, r1:n) = A(r1:m, r1:n) -
        (SPREAD (A(r1:m, r), 2, n-r) &
      * SPREAD (A(r, r1:n), 1, m-r))
```

   It can be expressed more simply using a FORALL:

```
FORALL (i = r1:m, j = r1:n) &
  A(i,j) = A(i,j) - A(i,r) * A(r,j)
```

3. To express subscript-dependent values. For example, the following sets each element `even(i,j)` of a logical array to .TRUE. if $(i + j)$ is even and .FALSE. otherwise:

```
FORALL (i=1:m, j=1:n) &
  even(i,j) = (MOD (i+j, 2) = = 0)
```

   Subscript-dependent expressions are very cumbersome to express in array syntax. The Fortran 90 equivalent of the above is:

```
even = (MOD(SPREAD((/(i,i=1,m/), 2, n) &
          + SPREAD((/(j,j=1,n/),1,m),2)
```

4. To express nonrectangular array sections:

```
FORALL (i=1:n) ... A(i,i) ...                          ! diagonal of array
FORALL (i=1:n, j=1:n, j >= i)  ... A(i,j) ... ! upper triangle
```

5. To express more general array access patterns. In fact, it is possible to select elements from an array in any fashion to form another array of any shape. For example:

```
FORALL (i=1:l, j=1:m, k=1:n) &
  ... A(ivec(i,j,k), jvec(i,j,k)) ...
```

forms a three-dimensional "irregular section" from the two-dimensional array A. Fortran 90 vector subscript notation could only form a one or two-dimensional section from A, in which `ivec` and `jvec` each depends on only one FORALL index (a different one for each dimension).

The next example expresses an array assignment whose right-hand side is a product of two $n \times n$ arrays, one formed from an array A by cyclically shifting each row i left by i places, the other formed from an array B by cyclically shifting each column j up by j places. This cannot be written as an array assignment, however, as this pattern of row and column shifts cannot be expressed by array sections. The subscript ranges are declared as $0:n-1$.

```
FORALL (i=0:n-1, j=0:n-1) &
  C(i,j)=A(i,MOD(i+j,n))*B(MOD(i+j,n),j)
```

If this is repeated n times, with the cyclic shifts increased by one each time, and the results are accumulated into C, the matrix product $C = AB$ is produced.

6. Finally, a FORALL statement must be used when the constituent elemental assignment involves a reference to a nonelemental function. For example, the following is a completely data parallel expression of the matrix multiplication $C = AB$, where A, B, and C have arbitrary sizes $(m \times k)$, $(k \times n)$, and $(m \times n)$, respectively:

```
FORALL (i=1:m, j=1:n) &
  C(i,j) = DOT_PRODUCT(A(i,:), B(:,j))
```

This cannot be written as an array assignment to the whole of C because of the reference to the nonelemental intrinsic function DOT_PRODUCT. Without FORALL, the assignment to `C(i,j)` would therefore have to be enclosed in DO-loops over i and j.

We shall see in the next section that FORALL assignments can also reference user-defined functions, subject to certain constraints.

A FORALL *construct* is also provided [HPF §4.2]. This allows a single FORALL header to govern a sequence of statements, which may be assignment statements, FORALL statements and constructs, and WHERE statements and constructs. Incidentally, FORALL index bounds and strides *can* depend on the FORALL indices of an enclosing FORALL construct. The FORALL construct is not included in Subset HPF, though the FORALL statement is. For completeness we mention that an assignment in a FORALL statement or construct may be a pointer assignment rather than a normal assignment.

6.2 PURE Procedures

The order of execution of the individual assignments in a FORALL statement is undefined—ideally they should all execute in parallel. Therefore, if a FORALL assignment contains a function reference, the function may be invoked concurrently for all FORALL index values. In addition to returning a value, an ordinary user-defined Fortran function can contain a variety of *side effects*, such as modifying dummy arguments or variables in common blocks, or performing I/O. Whenever such side effects can occur it is preferable that they should happen in a well-defined order, otherwise the net result may be nondeterministic. For example, if one function invocation writes to a variable that another reads, or two invocations write different values to the same variable, then the overall behavior depends on the order of the invocations. We have already indicated that a design objective of FORALL is that it should be deterministic, so this suggests that functions referenced in FORALL assignments should be side effect free.

Another consideration is implementation. In general, a function referenced in a FORALL assignment might be executed on a subset of the

processors that are allocated to the program, or even on a single processor; this would allow multiple references (for a set of FORALL instances) to be executed simultaneously on different processors. If the function can contain arbitrary data mapping directives, it might access variables stored in the local memories of processors that it is not executing on. This cannot be implemented on distributed memory architectures using pure message passing, as message passing requires that the processors at both ends of a communication execute the communication instruction. To support this behavior requires some degree of shared memory support, either in hardware or software.

For both of these reasons (but principally the first) it is forbidden to reference ordinary user-defined functions in a FORALL assignment or mask expression.#

However, HPF introduces a new class of functions called *pure* functions, which are guaranteed to be side effect free and which can be used in these contexts [HPF §4.3]. They are denoted by adding the keyword PURE before the FUNCTION keyword in the function header statement, and must satisfy a number of constraints, which are checkable at compile-time, to ensure that they are both side effect free and efficiently implementable under concurrent reference.

In outline, the constraints to ensure side effect freedom are as follows [HPF §4.3.1.1]. A pure function must not contain any operation that might conceivably change the value or pointer association of a dummy argument or global variable ([HPF p. 73, 3rd constraint] gives a full list of disallowed operations), or SAVE local variables, or reference nonpure procedures, or contain any external I/O, PAUSE, STOP, or dynamic remapping operations. Note the use of the word *conceivably* above; it is not sufficient for a function merely to be side effect free *in practice*. For example, a function that contains an assignment to a global variable but in a branch that is not executed is nevertheless not pure. This strictness is necessary to allow side effect freedom to be checked at compile-time. Data mapping is also restricted in a pure function, as we shall describe shortly.

Pure subroutines may also be defined, and must satisfy the same constraints except that they may modify their dummy arguments. They are useful for a variety of purposes, for example so that subroutines can be called from within pure functions, and so that FORALL assignments can be defined assignments, both of which require the use of a pure subroutine.

A pure procedure (i.e., function or subroutine) can be used anywhere that a normal procedure can. However, a procedure *must* be pure if it is used in any of the following contexts:

1. In a FORALL assignment or mask expression, or a statement in a FORALL construct
2. Within the body of a pure procedure
3. As an actual argument in a pure procedure reference

When a procedure is used in any of these contexts, its interface must be explicit, and both its interface and definition must specify the PURE keyword and the INTENT** of its nonpointer and nonprocedure dummy arguments (though admittedly this is redundant for a pure function as its arguments must be INTENT(IN) by definition). Intrinsic functions, including the new HPF intrinsic functions, are always pure and require no explicit declaration of this fact. Of the intrinsic subroutines, only MVBITS is pure; the others are not as they perform I/O. A statement function is pure if all functions that it references are pure. The PURE attribute is not included in Subset HPF.

Functional Parallelism

As an example of the use of pure functions, Figure 11 shows a program which plots the Mandelbröt set over a grid of points by calling a pure function `mandel` concurrently at every point from a FORALL statement. Note that, apart from prohibiting PAUSE and STOP statements, pure functions have no constraints on their internal control flow. Therefore, when referenced in a FORALL, they allow *functional parallelism* in an HPF program, as different concurrent invocations can execute different code.†† Thus in Figure 11, different invocations of `mandel` will execute different numbers of iterations of the WHILE loop, and some will execute the assignment in the IF statement while others do not. Apart from pure function references in FORALL, functional parallelism can also arise

However, the bound and stride expressions that define FORALL index ranges *can* reference normal functions (unless they are within an enclosing FORALL construct), as they are evaluated only once.

** Dummy arguments can be specified as INTENT(IN), (OUT), or (INOUT), meaning, respectively, that they are read, written, or both.

†† Of course, SIMD architectures cannot fully exploit this potential.

```
      REAL n (-100:50, -50:50)       ! #itns to diverge to (|z| > 2)
!HPF$ DISTRIBUTE n (BLOCK, BLOCK)
      INTERFACE
        PURE INTEGER FUNCTION mandel (c)
          COMPLEX, INTENT (IN) :: c
        END FUNCTION mandel
      END INTERFACE

      FORALL (i= -100:50, j= -50:50) n(i,j) = mandel (0.02*CMPLX (i,j))
      ...

    PURE INTEGER FUNCTION mandel (c)
      COMPLEX, INTENT (IN) :: c
      COMPLEX :: z
    !-------------------------------------------------------------!
    ! Returns the number of iterations for |z| to become > 2 under !
    ! z -> z**2 + c, starting at z = c.    If (|z| <= 2) after 100 !
    ! iterations it is assumed to remain so (i.e. 'c' is in the    !
    ! Mandelbrot set) and the special value -1 is returned.        !
    !-------------------------------------------------------------!
      z = c
      mandel = 0
      DO WHILE (ABS (z) <= 2.0  .AND. mandel < 100)
        z = z*z + c
        mandel = mandel + 1
      ENDDO
      IF (ABS (z) <= 2.0) mandel = -1
    END FUNCTION mandel
```

FIGURE 11 Using a PURE function to plot the Mandelbröt set.

via "independent" DO-loops and "extrinsic" procedure references, both of which are briefly introduced later.

Data Mapping in PURE *Procedures*

Data mapping is also restricted within pure procedures. The dummy arguments and result can be aligned among themselves, and local objects can be aligned among themselves or with the dummy arguments or result, but otherwise local and dummy objects may not be subject to any other type of mapping directives. The mapping of global variables is not constrained however.

These restrictions are imposed because multiple invocations of the procedure may be active simultaneously, each executing on a subset of the processors. As we have explained, on multiprocessor systems without shared memory support, the data accessed by a procedure must be contained in the local memories of the set of processors that are executing it. For efficiency the caller should have the freedom to choose the processor subset on which to execute any particular pure procedure reference, e.g., to maximize concurrency in a FORALL, and/or to reduce communication, taking into account the mappings of other terms in an expression or assignment. This implies that, on nonshared memory platforms, it must also have the freedom to map the procedure's actual arguments, result, and local variables to the chosen processor subset, just as it has this freedom generally for variables in an expression. Therefore, a dummy argument or result may not appear in any mapping directive that fixes its location with respect to the processor array. For example, it may not be aligned with a global variable or template, or be explicitly distributed, or even INHERIT its mapping, all of which would remove the caller's freedom to choose the actual's mapping. The only type of mapping information that may be specified for the dummy arguments and result is their alignment with each other, which may provide useful information to the caller

```
      INTEGER m, n, r, r1
      REAL    A (m,n)
!HPF$ DISTRIBUTE A (CYCLIC, CYCLIC)
      ...
      DO r = 1, m
         r1 = r+1
         A(r, r1:n) = A(r, r1:n) / A (r, r)
         FORALL (i=r1:m) A(i,r1:n) = update_row (A(i,r1:n), A(i,r), A(r,r1:n))
      END DO
      ...
      CONTAINS
        PURE FUNCTION update_row (row, factor, ref_row)
          REAL, INTENT (IN)  :: row (:),  factor,  ref_row (SIZE (row))
          REAL  :: update_row (SIZE (row))
!HPF$     ALIGN WITH row  :: ref_row,  update_row

          update_row = row - factor * ref_row
        END FUNCTION
```

FIGURE 12 Gaussian elimination with each row updated by a pure function call.

about their required *relative* mappings. For the same reasons, local variables may be aligned with the dummy arguments or result, but may not have arbitrary mappings.‡‡

This is not to say that the actual arguments of a pure procedure cannot be distributed. Indeed, they can have any mapping. The constraints simply restrict the *specification* of their mapping within the pure procedure, so the implementation can remap them as it sees fit. This is one place where the programmer is largely relieved of the burden of worrying about data mapping (expressions being another).

We can illustrate these points by considering one last version of the Gaussian elimination code, shown in Figure 12. This time each row of matrix A is updated by calling a pure function update_row, and this is done in parallel over all the rows in a FORALL statement. A is distributed cyclically over a two-dimensional processor array. (Incidentally, in this example update_row is a Fortran 90 *internal* function whose interface is automatically explicit in the caller. Internal functions are not included in Subset HPF.)

An efficient implementation of the FORALL might broadcast row A(r, r1:n) so that it is aligned with every row A(i, r1:n), $i > r$, according to the alignment specified in the pure function,§§ and then execute each instance i of the FORALL on the processors that own the relevant assignment variable (and argument) A(i, r1:n), namely, on a subset of one row of the processor array. Therefore different rows of processors will update different rows of A in parallel, and multiple invocations of update_row will be active simultaneously.

This implementation might easily be ruled out if the programmer could specify arbitrary mappings for update_row's arguments and local variables. For instance, if "INHERIT ref_row" were specified, then strictly speaking it would prevent the corresponding actual argument A(r, r1:n) from being broadcast, so every invocation of update_row would have to be activated on the same subset of processors—namely those owning A(r, r1:n)—thus sequentializing the FORALL instances.

In general each individual invocation of update_row is distributed across multiple processors—namely the row of processors owning the argument A(i, r1:n)—so update_row exploits parallelism both internally and via concurrent reference. Since a pure function may be executed on multiple processors, it is useful to be able to specify how its arguments should be aligned relative to each other. This enables the caller to map them in

‡‡ However, the implementation of nonshared memory platforms is still complicated by the fact that pure procedures can access common block and module variables whose mapping *is* fixed with respect to the processor array.

§§ The caller is aware of the dummy argument mapping specified in pure function update_row because its interface is explicit, as it must be when a function is referenced in a FORALL.

a manner that is efficient for the operations performed within the function.

6.3 INDEPENDENT Directive

HPF also introduces an INDEPENDENT directive, which can precede a DO-loop or FORALL statement or construct [HPF §4.4].

If it precedes a DO-loop it asserts that the loop iterations are *independent*, meaning that they can be executed in any order, and therefore concurrently, without changing the semantics of the loop. The conditions that must be satisfied for this to apply are listed in [HPF pp. 81–82]. Unlike the case for PURE procedures, these are assertions about *behavior*, and do not imply any syntactic constraints. The DO-loop may contain procedure calls, branches in control flow, etc., so different iterations may execute different code, giving scope for functional parallelism. An example is:

```
!HPF$ INDEPENDENT
      DO i=1, 100
        a (p(i)) = b (i)
      ENDDO
```

which asserts that p(1:100) does not contain any repeated entries (otherwise the same element of a would be assigned by more than one iteration and the result would depend on their execution order). This is therefore equivalent to the array assignment:

```
a (p(1:100)) = b (1:100)
```

which implies the same condition on p.

When it precedes a DO-loop, the INDEPENDENT directive also has an optional NEW clause to specify that certain variables must be regarded as private to each iteration in order to make the iterations independent. That is, each iteration must be given a new, independent copy of the variable which is undefined at the start of the iteration and becomes undefined again at the end. This clause is only valid if this modification does not change the meaning of the program, i.e., if the private variables do not carry values from one iteration to another, or into or out of the loop.

We should point out that, except in simple cases, the iterations of an independent DO-loop may only be concurrently executable on shared memory MIMD machines. (Indeed, this particular feature has its origin in Fortran dialects for such machines.) This is because of the complete generality of data references allowed within them, which may inhibit concurrent execution on pure message-passing systems, and of control flow, which may prevent concurrent execution on SIMD machines. Therefore, if a program is intended to be run on nonshared memory achitectures, we recommend the use of array or FORALL syntax rather than independent DO-loops whenever possible.

If it precedes a FORALL statement or construct, the INDEPENDENT directive asserts that the variable(s) written for one combination of FORALL indices are not referenced (i.e., read or written) for any other combination of FORALL indices. For example:

```
!HPF$ INDEPENDENT
      FORALL (i=1:m) a (i) = a (i+n)
```

asserts that the array sections a(1:m) and a(1+n:m+n) are either equivalent (i.e., $n = 0$) or completely disjoint (i.e., $n \leq -m$, or $n \geq m$). This condition means that the various synchronization points implicit in a FORALL's semantics—namely between evaluating the right-hand sides and performing the assignments of an assignment statement, and between successive statements in a FORALL construct—are unnecessary and can be removed. In particular this means that FORALL assignments can proceed directly rather than via temporary intermediate storage, which is a useful optimization.

As with all directives that provide information about program behavior, the INDEPENDENT directive should only be used to assert actual behavior and not to try to change that behavior. If the information asserted by the directive is incorrect then the program is erroneous and its behavior is undefined.

7 OTHER HPF EXTENSIONS

HPF includes a number of other extensions which we summarize here. We do not describe them in detail due to lack of space, but instead indicate where full details can be found in the "High Performance Fortran Language Specification."

HPF introduces three new intrinsic functions. They are the *system enquiry* intrinsic functions NUMBER_OF_PROCESSORS and PROCESSORS_SHAPE [HPF §5.2, 5.6.4, 5.6.5], which were introduced in Section 3.1, and a new computational intrinsic function ILEN [HPF §5.6.1]. It also ex-

tends the Fortran 90 intrinsic functions MINLOC and MAXLOC by giving them an extra optional argument DIM for finding the locations of the maximum and minimum elements along a given dimension [HPF §5.6.2, 5.6.3]. In Subset HPF this argument, if specified, must be an initialization expression.

HPF defines a *standard library* of procedures in a module called HPF_LIBRARY [HPF §5.4 and 5.7]. It contains:

1. Subroutines for enquiring about data mapping: HPF_ALIGNMENT, HPF_TEMPLATE, and HPF_DISTRIBUTION
2. New array reduction functions IALL, IANY, IPARITY, and PARITY, which apply the operators IAND, IOR, IEOR (i.e., bitwise AND, OR, EOR), and .NEQV. (logical EOR), respectively
3. Array "combining scatter" functions XXX_SCATTER, and "parallel prefix" and "suffix functions" XXX_PREFIX and XXX_SUFFIX, where XXX is any of the available reduction operations
4. Array sorting functions: GRADE_UP and GRADE_DOWN
5. Bit manipulation functions: LEADZ, POPCNT, and POPPAR

This module is not included in Subset HPF.

It is possible to escape from HPF to another programming model and/or language by calling non-HPF procedures called *extrinsic procedures* [HPF §6]. Their interface must be explicit and must specify "EXTRINSIC (*model-name*)" in the procedure header statement, where *model-name* is the name of an implementation-dependent programming model or language. For example, on a distributed memory MIMD machine this might allow an HPF program to invoke message-passing code in order to obtain forms of MIMD parallelism that cannot be achieved in HPF, or to hand-tune critical kernels, at the expense of nonportability. The EXTRINSIC mechanism is not included in Subset HPF.

HPF defines one particular type of extrinsic model, called "HPF_LOCAL" [HPF Annex A]. This is basically Fortran 90 operating on the local data on each processor, together with a library of procedures for relating the local and HPF views of data and enquiring about abstract processor coordinates. However, this is an optional part of the standard, as it may not be implementable on SIMD architectures.

Finally, all variables and common blocks that are subject to sequence and/or storage association *must* be identified by a SEQUENCE directive. This is the only respect in which a standard-conforming Fortran 90 program is *not* a standard-conforming HPF program unless it is modified. These associations imply restrictions on the mapping of the variables concerned [HPF §7].

8 DISCUSSION AND CONCLUSIONS

HPF has some obvious advantages over explicit SPMD programming with message passing. It is closer to the style of programming familiar to ordinary Fortran programmers and offers a relatively simple migration path for existing Fortran codes. Because HPF programs are not cluttered with message-passing details they are shorter, clearer, and easier to develop and modify than their message-passing equivalents. The performance of an SPMD program depends critically on its data mapping, and it is easier to experiment with different data mappings by changing the directives in an HPF program than by recoding a message-passing program.

Furthermore, this higher-level programming style does not necessarily incur lower performance, because by and large HPF has been designed to permit direct message-passing implementation on distributed memory systems, generally avoiding the overheads of simulated shared memory. Indeed, it is arguable that in the long term HPF can actually be more efficient than explicit message-passing programming, because an HPF compiler can directly target low-level machine instructions for communications rather than going through message-passing portability layers. It can also employ optimization techniques such as overlap areas, code reordering, and message vectorization and coalescing, that the programmer may not have the expertise or inclination to use. Further research and development in HPF compilation will doubtless improve performance further.

Having said this, writing efficient HPF programs will not necessarily be a trivial task. Indeed, the high-level nature of the language means that it will be very easy to write hugely inefficient code. Deceptively simple operations can translate into code involving enormous amounts of communication. The programmer will need a good understanding of the program and of the meaning of HPF's mapping directives (which we hope this ar-

ticle has helped to impart) in order to map data effectively. In addition, we anticipate that some old "dusty deck" Fortran programs may need to be significantly rewritten to convert them to efficient HPF programs, in particular making use of some of the new features of Fortran 90 and HPF, e.g., using array and FORALL syntax rather than DO-loops where possible, removing sequence and storage associations, etc. Fortunately all of these optimizations are "clean," in that they should improve code legibility as well as efficiency.

ACKNOWLEDGMENTS

We would like to thank Bryan Carpenter for providing the Gaussian elimination example, John Eastmond for critically reading the manuscript, and Jerry Wagener for providing information and references about Fortran 95.

The authors gratefully acknowledge support from the Engineering and Physical Sciences Research Council under grant number GR/J89507.

REFERENCES

[1] High Performance Fortran Forum, "High Performance Fortran language specification," *Sci. Prog.*, Vol. 2, pp. 1–170, 1993.

[2] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, Jr., and M. E. Zosel, *The High Performance Fortran Handbook.* MIT Press, Cambridge, 1994.

[3] B. Chapman, P. Mehrotra, and H. Zima. "Programming in Vienna Fortran," *Sci. Prog.*, Vol. 1, pp. 31–50, 1992.

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

[5] C. Koelbel and P. Mehrotra, "Compiling global name-space parallel loops for distributed execution," *IEEE Trans. Parallel Distrib. Systems.* Vol. 2, pp. 440–451, 1991.

[6] Pacific Sierra Research Corp., *MIMDizer User's Guide, version 7.02.* Placerville, CA: Pacific Sierra Research Corporation, 1991.

[7] G. Fox, S. Hirananadani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng and M-Y. Wu. "Fortran D language specifiation," Technical Reports COMP TR90-141, Department of Computer Science, Rice University, Houston, TX, December 1990, and SCCS-42c, Syracuse Center for Computer Science, Syracuse University, Syracuse, NY, April 1991.

[8] M-Y. Wu and G. Fox, "Fortran 90D compiler for distributed memory MIMD parallel computers," Technical Report SCCS-88b, Syracuse Center for Computer Science, Syracuse University, Syracuse, NY, July 1991.

[9] J. H. Merlin, "ADAPTing Fortran 90 array programs for distributed memory architectures," *Proc. 1st International Conference of the ACPC,* Salzburg, October 1991.

[10] J. H. Merlin. "Techniques for the automatic parallelisation of Distributed Fortran 90," Technical Report SNARC 92-02, Department of Electronics and Computer Science, University of Southampton, November 1991.

[11] F. André, J.-L. Pazat, and H. Thomas, *Proc. of 1990 ACM International Conference on Supercomputing*, 1990, Amsterdam, Netherlands, pp. 380–388.

[12] ANSI X3J3 document 94-009 (revision 2), 11 July 1994. Available by anonymous ftp from the `x3j3` directory at `ftp.ncsa.uiuc.edu` or `ftp.dfrf.nasa.gov`.