

High Performance Fortran

John Merlin,
Department of Electronics and Computer Science,
University of Southampton,
Southampton, S09 5NH, U.K.
(jhm@ecs.soton.ac.uk)

June 1, 1993

Abstract

High Performance Fortran 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 expressing data distribution across multiple memories, and concurrent execution features. This paper provides an informal introduction to the main features of HPF.

1 Introduction

High Performance Fortran (HPF) is a new informal language standard which promises to greatly simplify the task of programming data parallel applications for distributed memory MIMD machines.

It is generally accepted that the largest obstacle to the widespread use of distributed memory message-passing systems is the difficulty currently encountered in programming them. 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.

HPF aims to remove this burden from the programmer. It comprises a set of extensions to Fortran 90. The central idea of HPF is to augment a standard Fortran program with directives that specify the distribution of data across disjoint memories. The HPF compiler then handles the messy business of partitioning the data according to the data distribution directives, allocating computation to processors according to the locality of the data references involved, and inserting any necessary data communications in an implementation dependent way, e.g. by message-passing or by a (possibly virtual) shared memory mechanism.

HPF is also designed to be largely architecture independent. It can be implemented across the whole spectrum of multi-processor architectures: distributed and shared memory MIMD, SIMD, vector, workstation networks, etc, and can even be implemented on single-processor systems, because data distribution is specified by means of *directives*, i.e. structured comments which do not affect the program semantics and are significant only to an HPF compiler. (Having said this, it is not true that a general HPF program can be compiled by a standard Fortran 90 compiler, as HPF contains a few actual *language* extensions to

Fortran 90, as we describe later. However, an HPF program that avoids these language extensions can indeed be compiled as a standard Fortran program). Thus HPF aims to solve the dual problems of the difficulty of programming distributed memory systems and the lack of portability of the resulting programs.

The language was developed by the ‘High Performance Fortran Forum’, a working group convened in January 1992 by Ken Kennedy of Rice University and Geoffrey Fox of Syracuse University. The group included representatives of most parallel computer manufacturers, several compiler vendors, and a number of government and academic research groups in the field of parallel computation. It held meetings every 6 weeks in 1992 and twice in 1993, the last time in March. This activity is now finished, but a further development of the language, ‘HPF-2’, is planned for 1994 to consider additional features, such as support for irregular data distributions and more general MIMD parallelism.

The formal language definition is contained in the ‘High Performance Fortran Language Specification’, [HPFF, 1993a] (the final editing of which, incidentally, was completed during the preparation of this paper, on 21 May to be precise!). This document also defines an official subset of the language, ‘Subset HPF’, to facilitate early implementation.

A number of features that were considered but not accepted into HPF are presented in a separate document, the ‘HPF Journal of Development’ [HPFF, 1993b]. These features were rejected for lack of time or consensus, or in order to minimise direct extensions to Fortran 90, rather than because of technical flaws, and are therefore documented so that they are available for consideration in future language design activities, including HPF-2.

This paper provides an informal introduction to the main features of HPF. Section 2 describes the extensions for specifying data distribution. Section 3 presents the extra facilities provided by HPF for expressing data parallel and concurrent execution. Sections 4 and 5 summarise the remaining HPF extensions and Subset HPF respectively.

2 Data mapping

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

1. Data objects (i.e. scalars or arrays) may be *aligned* with other data objects or *templates* (special virtual objects that occupy no storage, which are described and motivated further below). This sets up a relation between the elements of two 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 the object that it is aligned with (namely *B*) is called the *align target*.
2. Templates or data objects that are not alignees are *distributed* over *abstract processors*. The distribution of an align target also determines that of all the objects that are aligned to it.

Typically, in multi-processor architectures with distributed memory, a processor can access data much faster from its ‘local’ memory than from other memories; the latter requires the non-local data to be ‘communicated’. Therefore, the motivation for specifying alignment is that an operation on two or more data objects is likely to be executed much faster if the data objects are ‘aligned’, or stored in the same memory, as it can be performed without communications by the processor that stores the data objects 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 minimise communication, while keeping elements that can be operated on independently apart to maximise data parallelism. It is clearer and safer to be able to specify alignment explicitly rather than relying on it being achieved as a side-effect of distribution.

Another reason for this two-level mapping is that the optimal alignment is often largely determined by the program, while the optimal distribution is often largely determined by the target architecture, so tuning a program for different architectures would mainly involve changing the distribution.

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 (e.g. the former may actually be ‘processes’, and an implementation may be able to execute multiple processes concurrently on each physical processor).

Data mapping in HPF is specified by *directives*. These are structured Fortran comments that are distinguished by starting with the characters ‘**HPF\$**’ immediately after the comment character (i.e. immediately after ‘!’ in free source form, or ‘C’, ‘*’ or ‘!’ in column 1 in fixed source form). Being structured comments they are ignored by a standard Fortran compiler and only recognised by an HPF compiler. 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 this is not defined by the language, e.g. the order of the operations in an intrinsic reduction function like **SUM**, or of the elemental expression evaluations and assignments that comprise an array assignment.

This section describes data mapping in HPF.

2.1 Templates

HPF introduces the concept of a *template*, which is a virtual scalar or array, i.e. one that occupies no storage. Templates are declared by a **TEMPLATE** directive, e.g.:

```
!HPF$ TEMPLATE S, T (16), U (2*N+1, 2*N+1)
```

The sole function of a template is to provide an abstract object with which data objects can be aligned and which can be distributed—i.e. it provides an intermediary in the mapping of data objects to abstract processors. It is not mandatory to use templates in this rôle—data objects can be aligned directly with other data objects, and can also be distributed directly. However, the ability to declare and use templates has stylistic advantages and turns out to be invaluable in some situations, as we shall explain later in this section.

We shall use templates as intermediaries in many of the examples in this section.

2.2 Alignment

The **ALIGN** directive relates the elements of a data array to the elements of another data array or a template, such that elements that are ‘aligned’ with each other are guaranteed to be mapped to the same processor(s), regardless of the distribution directives. Alignment can also be specified for scalar objects. However, for most of this section we shall concentrate on describing the alignment of array objects, and mention briefly at the end how scalars may be aligned (which should be obvious anyway by that stage).

For example, given 2-dimensional arrays **A** and **B** with the same shape:

```
!HPF$ ALIGN A (:,:) WITH B (:,:)
```

declares that each element of **A** is ‘aligned’ with the corresponding element of **B** (i.e. an identity alignment). This ensures that, for any values of **i** and **j**, element **A(i,j)** will be mapped to the same processor(s) as element **B(i,j)**. 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, and 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.

The array immediately after the **ALIGN** keyword is called the *alignee*, and the array or template after the **WITH** keyword, with which it is aligned, is the *align target*. All elements of the alignee are involved in the alignment relation. In the **ALIGN** directive, the alignee name is followed by a parenthesised list with an entry for each dimension which must be one of three things:

- a colon ‘:’,
- a scalar integer named variable called an *align dummy*, or
- an asterisk ‘*’.

We will consider these three cases in turn.

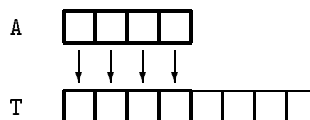
The simplest case is when all dimension entries for the alignee are ‘:’s, which is a standard Fortran 90 way of specifying a whole array. (Unfortunately, the Fortran 90 shorthand for specifying a whole array by just giving its name cannot be used in the form of **ALIGN** directive that we shall describe here.) The align target is then in general a *regular section* of an array or template, specified using the normal Fortran 90 subscript triplet notation.¹ Alignee dimensions containing a ‘:’ are matched in order with target dimensions containing subscript triplets, and matching dimensions must have the same number of elements selected.² Each element of the alignee is aligned with the corresponding element of the align target (i.e. that in the same position within the regular section).

E.g., with the declarations:

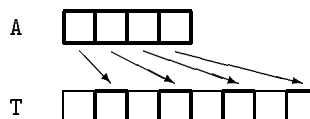
```
REAL A(4)
!HPF$ TEMPLATE T(8)
```

some possible alignments are:

```
!HPF$ ALIGN A (:) WITH T (1:4)
```



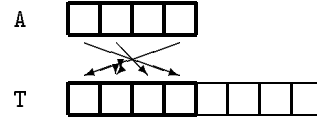
```
!HPF$ ALIGN A (:) WITH T (2:8:2)
```



¹ A subscript triplet has the general form $[l]:[u]:[s]$, where l , u and s are scalar integer expressions and $[\dots]$ denotes an optional item. It denotes the subset of elements with subscripts from l to u in steps of s , the stride. If l or u is omitted it defaults to the declared lower or upper bound of the dimension respectively, and if s is omitted it defaults to 1; thus a solitary ‘:’ is a special case of a subscript triplet that specifies a whole dimension.

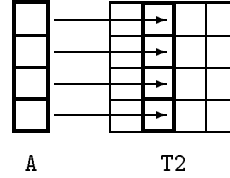
² In Fortran 90 terminology, the alignee and align targets must *conform*, or have the same shape.

```
!HPF$ ALIGN A (:) WITH T (4:1:-1)
```



As normal for Fortran 90 regular sections, any dimension of the align target can contain a scalar subscript rather than a subscript triplet. The only requirement is that the regular section selected from the align target conforms with the alignee.³ In this way an array can be ‘embedded’ within a larger dimensional array or template, as in the following example:

```
!HPF$ ALIGN A (:) WITH T2 (:,2)
```



An alternative to the subscript triplet notation is the ‘align dummy’ notation, whereby a scalar integer named variable, called an *align dummy*, appears in a dimension of the alignee, and an expression that is linear in the align dummy appears in one dimension of the align target. Align dummies in different alignee dimensions must be distinct. Dimensions of the alignee and align target involving the same align dummy are then matched. Matching dimensions of the alignee and target need not be in the same order, unlike the case for subscript triplet notation. An element of the alignee with subscript value s in the dimension concerned is then aligned with an element of the align target whose subscript value in the matching dimension is $f(s)$, where $f(d)$ is the linear expression in the align dummy d . Obviously $f(s)$ must be a valid subscript for the target dimension for all s in the subscript range of the alignee dimension. Notice that, because $f(s)$ is linear in s , it generates a regular section when applied to the complete subscript range of s , so in this sense it is equivalent to the subscript triplet notation. E.g., with the same declarations as before (`REAL A(4)` and `TEMPLATE T(8)`):

```
!HPF$ ALIGN A (I) WITH T (2*I - 1)
```

has the same meaning as:

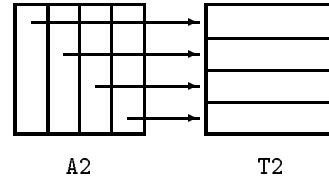
```
!HPF$ ALIGN A (:) WITH T (1:7:2)
```

(since I takes subscript values in the range $[1:4]$).

Because this form does not require matching dimensions of the alignee and target to appear in the same order, it can be used to *permute* dimensions in the alignment mapping. This cannot be achieved using subscript triplets alone, as alignee dimensions specified by colons are matched with subscript triplets in the align target in order of appearance. E.g., an array A2 can be aligned with the *transpose* of a template T2 using the following directive:

```
!HPF$ ALIGN A2 (I,J) WITH T2 (J,I)
```

(*dimensional permutation*)



³As usual in Fortran 90, scalar subscripts are ignored for the purpose of determining the shape of a regular section.

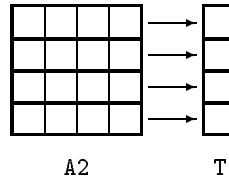
(This is a good alignment if, for example, another array **B** has an identity alignment with **T2**, and element **A2**(*i*,*j*) often appears together with **B**(*j*,*i*) in expressions and assignments.) Indeed, dimensional permutation is the only real justification for using the align dummy notation—otherwise, the triplet notation is clearer and more succinct. Both forms can be used in the same directive, so the use of align dummy notation can be restricted to just those dimensions necessary for dimensional permutation. E.g. the above could equally well be written:

```
!HPF$ ALIGN  A2 (I,:)  WITH  T2 (:,I)
```

A ‘*’ may appear as a dimension entry in any dimension of the alignee or target, and it means that the subscript in that dimension plays no part in the alignment relation. Thus, if a ‘*’ appears in an *alignee* dimension, it means that each set of elements whose subscripts differ only in that dimension is aligned to the same element(s) of the align target (which is called *collapsing* a dimension). For example:

```
!HPF$ ALIGN  A2 (:,*)  WITH  T  (:)
```

(2nd dimension of ‘A2’ collapsed)

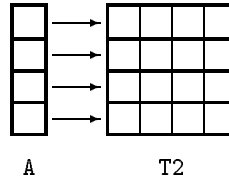


This alignment means that every element in a given row of **A** is mapped to the same processor(s) (since each row is aligned to a single element of **T**, which cannot be split across multiple processors however **T** is distributed). This means that operations and assignments involving different elements in the same row will be executed without communications, at the expense of preventing concurrent operations and assignments on the elements of a row.

If a ‘*’ appears in the *align target*, it means that each element of the alignee is aligned with a whole set of elements of the target whose subscripts differ only in that dimension (i.e. the alignee is copied, or *replicated*, over that dimension of the align target). For example:

```
!HPF$ ALIGN  A  (:)  WITH  T2 (:,*)
```

(‘A’ replicated over 2nd dimension of ‘T2’)



causes **A** to be replicated over the second dimension of **T**, and consequently also replicated over whatever processor array dimension the second dimension of **T** is distributed over.⁴ Replicating a variable has the advantage that its value can be read by multiple processors without communication, but the disadvantage of complicating its updating, as all copies must be updated. This is necessary because all copies of a replicated variable 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. The storage overhead of keeping multiple copies may also be a disadvantage, particularly for large data objects. Nonetheless, this is a natural and sensible storage strategy for scalar variables, particularly control variables such as **DO**-loop indices. It is also a natural strategy for mapping

⁴Incidentally, the alignment notation should not be taken too literally in the case of replication. In this example, if the second dimension of **T2** were distributed over 2 processors, the alignment directive suggests that each would store 2 identical copies of **A**. There is an obvious optimisation!

arrays onto a higher dimensional processor array, and may well be a good distribution scheme for small arrays that are read more frequently than they are written (e.g. small ‘lookup’ tables).

Collapsing and replication may of course be combined. Thus:

```
!HPF$ ALIGN A (*) WITH T (*)
```

means that all elements of **A** are aligned with every element of **T** (i.e. **A** is collapsed and then replicated over **T**). This means that every processor over which **T** is distributed will store a complete copy of **A**.

Notice that only *linear* alignments (i.e. aligning arrays to regular sections) can be specified. Irregular alignments (i.e. aligning arrays to irregular sections specified by vector subscripts) are not allowed, and neither are *skew* alignments, whereby several dimensions are aligned to a single dimension or vice versa.⁵

So far we have concentrated on aligning array objects. Scalar data objects and templates can also be involved in an alignment relation in ways which follow straightforwardly from the above discussion: a scalar can be aligned with another scalar or with an array element (i.e. it can be ‘embedded’ into an array), or be totally replicated over an array, or be aligned with an array target that has scalar subscripts in some dimensions and ‘*’s in others (a combination of embedding and replication); and an array can be totally collapsed onto a scalar object.⁶

A data object cannot appear more than once as an alignee in an **ALIGN** directive in a given scoping unit—once it has been aligned it cannot be aligned again.

Notice that, by aligning data objects to other data objects rather than templates, it is possible to create an ‘alignment tree’ as in Figure 1. Naturally, the alignment relation is transitive, i.e. if an element *a* of array *A* is aligned with element *b* of array *B*, which is turn aligned with element *c* of array *C*, then *a* is also aligned with *c*. All data objects in an alignment tree are said to be *ultimately aligned* with the object at the ‘root’ of the tree. In fact, all data objects in the tree are regarded as being directly aligned with the root, intermediate objects serving only for convenience in describing the alignment. The avoidance of such alignment trees, which can obfuscate alignment relations, is one advantage of aligning data objects to templates.

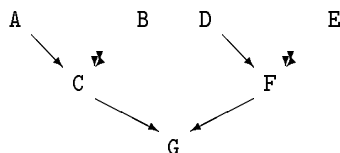


Figure 1: An ‘alignment tree’

Finally, we should emphasise that the alignment relation is directed: it is in general *not* symmetric. An array can be aligned with a *subset* of another array, but not vice versa. That is why we have been careful to distinguish the rôles of alignee and align target, and why an alignment tree can be constructed and has a definite root.

⁵Some of these restrictions may be relaxed in ‘HPF-2’ in 1994.

⁶However, if the *alignee* is scalar a different form of **ALIGN** directive to that described here must be used. This quirk arises from the desire to ensure that HPF directives are unambiguous if blanks are insignificant, as they are in Fortran 90 fixed source form.

2.3 Processors

The `PROCESSORS` directive declares abstract processor arrangements. For example:

```
!HPF$ PROCESSORS SCALAR_PROC, P(4), Q(8, NUMBER_OF_PROCESSORS()/8)
```

declare a scalar (i.e. single) abstract processor and two abstract processor arrays of the given dimensions. Incidentally, '`NUMBER_OF_PROCESSORS()`' is an HPF intrinsic function that returns the number of *physical* processors on which the program is executing. (Section 4 contains a brief survey of the new intrinsic functions introduced by HPF.)

Abstract processor arrays with different shapes may be declared in the same procedure, 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 arrangements 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.

2.4 Distribution

The `DISTRIBUTE` directive 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 the 'root' of an alignment tree or an object that has not appeared in an `ALIGN` directive can be distributed. Templates can always be distributed, as they are never alignees.

The distribution that can be specified for scalar objects is somewhat limited: they can only be distributed onto scalar processors.⁷ Therefore we shall forget scalar object for the rest of this section and concentrate on the distribution of array data objects and templates.

In that case, a so-called *distribution format* is specified for each dimension of the *distributee* (i.e. the object that is distributed). Only four distribution formats are provided: *block*, *cyclic*, *block-cyclic* or *collapsed* (i.e. undistributed). For simplicity we shall illustrate them for a 1-dimensional template:

```
!HPF$ TEMPLATE T (12)
```

distributed over a 1-dimensional processor array:

```
!HPF$ PROCESSORS P (4)
```

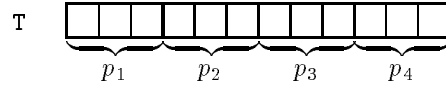
Block distribution. A *block* distribution means that the elements are divided into uniform (or nearly uniform) blocks of consecutive elements, the first of which is allocated to the first processor, the second to the second processor, etc. If the number of elements, n , is exactly divisible by the number of processors, p , then the blocks are of uniform size n/p . Otherwise, the blocks are made as uniform as possible; to be precise, blocks of size $b = \lceil n/p \rceil$ are allocated to the first $\lfloor n/b \rfloor$ processors, the remaining $n \setminus p$ elements form a small block which is allocated to the next processor, and no elements are allocated to any remaining processors. Note that the elements never 'wrap around' the processor array in a block distribution.

For example:

⁷To replicate a scalar over a processor array, it must be given a replicated alignment with an *array* object (i.e. data array or template) which is then distributed over the processor array. We should also mention that a different form of `DISTRIBUTE` directive to that described here must be used for scalar objects, to ensure unambiguity in the presence of insignificant blanks.


```
!HPF$ DISTRIBUTE T (BLOCK) ONTO P
```

results in the elements being allocated to processors as follows:



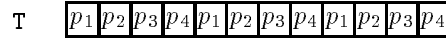
An explicit blocksize can be specified in parentheses after the **BLOCK** keyword. (By definition, **BLOCK** means **BLOCK**($\lceil n/p \rceil$) as we have said.) Thus in the above example, **BLOCK** (4) would mean that the elements are distributed in blocks of 4 rather than the default value of 3 (in which case only processors **P**(1) to **P**(3) would be allocated elements; **P**(4) would be unoccupied.) The specified blocksize must be such that the elements do not ‘wrap around’ the processor array. Thus in this example the blocksize must be ≥ 3 , as a blocksize of 1 or 2 would cause ‘wrap-around’ and be erroneous. To allow ‘wrap around’ a *cyclic* or *block-cyclic* distribution must be specified, as we shall now describe.

Cyclic and block-cyclic distribution. In the basic type of cyclic distribution, the first element is allocated to the first processor, the second to the second processor, etc. If there are more elements than processors the distribution ‘wraps around’ the processor array cyclically until all the elements are allocated. E.g. if n elements are cyclically distributed over $p < n$ processors, element p is allocated to processor p , element $(p+1)$ to processor 1, element $(p+2)$ to processor 2, etc.

For example:

```
!HPF$ DISTRIBUTE T (CYCLIC) ONTO P
```

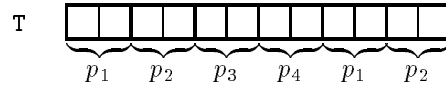
results in the following allocation of elements to processors:



An explicit blocksize may be specified in parentheses after the **CYCLIC** keyword, just 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. If they do, the distribution is often called *block-cyclic*. For example,

```
!HPF$ DISTRIBUTE T (CYCLIC (2)) ONTO P
```

results in the following distribution of elements to processors:



Incidentally, notice that **BLOCK**(b) and **CYCLIC**(b) are identical when there is no wrap around (i.e. when $n \leq bp$).

Cyclic distributions are sometimes 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, LU decomposition, is given later.

Collapsed distribution. An asterisk in a distributee dimension means that it is *collapsed*, i.e. *not* distributed. If this is applied to our one-dimensional template **T**, the target processor arrangement must be scalar:

```
!HPF$ DISTRIBUTE T (*) ONTO SCALAR_PROC
```

These descriptions generalise straightforwardly to multi-dimensional 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 multi-dimensional distributee. Each dimension is either distributed over a processor array dimension or is collapsed. ‘Skew’ distributions, whereby a single array or template dimension is distributed over several processor array dimensions, or vice versa, cannot be described. Therefore, the number of **BLOCK** and **CYCLIC** entries (with or without a blocksize) must equal the number of processor array dimensions, and the *n*th distributee dimension with such an entry is distributed over the *n*th processor array dimension. E.g.:

```
!HPF$ TEMPLATE U (10,10,10,10)
!HPF$ PROCESSORS Q (4,4)
!HPF$ DISTRIBUTE U (BLOCK(4), *, CYCLIC, *) ONTO Q
```

means that the first dimension of **U** is distributed **BLOCK(4)** over the first dimension of **Q**, the third dimension of **U** is distributed cyclically over the second dimension of **Q**, and dimensions 2 and 4 of **U** are collapsed (i.e. for fixed subscripts in the other dimensions, all subscript values in dimensions 2 and 4 are mapped to the same processor).

Examples of distributing a 2-dimensional template **T2** onto processor arrays **P(4)** and **Q(2,2)** are as follows:

```
!HPF$ DISTRIBUTE T2 (BLOCK, *) ONTO P
```

p_1
p_2
p_3
p_4

```
!HPF$ DISTRIBUTE T2 (*, BLOCK) ONTO P
```

p_1	p_2	p_3	p_4
-------	-------	-------	-------

```
!HPF$ DISTRIBUTE T2 (BLOCK, BLOCK) ONTO Q
```

$q_{1,1}$	$q_{1,2}$
$q_{2,1}$	$q_{2,2}$

The **ONTO** clause may be omitted from the **DISTRIBUTE** directive, in which case the distribution is onto an implementation-dependent processors arrangement. (Although the HPF specification says nothing on this point, it is conceivable that some implementations may allow a default processors 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 scoping unit to provide the default.)

2.5 Data mapping across procedure boundaries

So far we have only considered the mapping of variables that have their own storage, such as local and global variables. The situation is more complicated for a *dummy argument*, as it does not necessarily receive fresh storage, but instead provides a name for a virtual object that is associated with a number of other objects, the *actual arguments*, during program execution.

HPF actually provides a number of options for describing the mapping of a dummy argument. It can be given a *prescriptive* mapping, which forces the actual to acquire the specified mapping, or a *descriptive* mapping, which asserts that the actual will already be mapped as described, or it can *inherit* its mapping from the actual. In fact, the mapping can be described using any combination of these forms. We shall now describe these three possibilities in turn, and finally consider argument mapping from the caller's viewpoint.

2.5.1 Prescriptive mapping

The mapping of a dummy argument can be specified in the same way as for other data objects, using the directives described above. This constitutes a *command* to make the associated actual argument have the described mapping, and is called *prescriptive mapping*.

If the actual argument is not mapped as prescribed, it is automatically remapped on entry to the procedure to satisfy the dummy's mapping directives, and its original mapping is restored on return (unless this is known to be unnecessary, e.g. the actual argument is an expression and so no value is returned). In other words, the mapping of data objects may change at procedure boundaries if prescriptive directives are used.

There are several reasons why this facility is desirable and indeed necessary:

- Different invocations of a given procedure may in general receive different arguments with different data mappings.
- 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.
- 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.

2.5.2 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 (2:8:2)
```

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

```
!HPF$ DISTRIBUTE D *(BLOCK) ONTO *P
```

(where **D** is a dummy argument name in both examples). Clauses preceded by an asterisk are called *descriptive*, and constitute an *assertion* that the actual argument associated with that particular dummy argument will already have the described mapping characteristics.

No runtime checking or remapping will be performed within the procedure to satisfy these descriptive clauses. If the actual does *not* have the described mapping, the program is erroneous and its behaviour is undefined.

(But incidentally, runtime checking and remapping will be performed by the *caller* to satisfy these mapping directives, if the interface of the called procedure is explicit and specifies the dummy argument mapping directives. This is described later).

2.5.3 Transcriptive (or inherited) mapping

The dummy argument mapping can be ‘inherited’ from the actual argument using the **INHERIT** directive, e.g.:

```
!HPF$ INHERIT D
```

If no other information is provided about the dummy, this means that it can be associated with an actual with *any* mapping, and the actual argument will not be remapped—even if it is an array element or regular section. In general, code will be generated to handle *any* mapping for the argument (unless the compilation system can perform analysis to determine the possible actual argument mappings).

This basic idea of inherited mapping, as described above, 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 mapping and prescribing or describing others, which we shall now describe. These ‘mixed’ specifications, as well as being complicated, are probably of limited usefulness. (The reader is encouraged to skip the next 2 paragraphs if he wishes to avoid confusion!)

A dummy for which **INHERIT** is specified may optionally also appear in a **DISTRIBUTE** directive, though not in an **ALIGN** directive. In this case, the **DISTRIBUTE** directive provides information about the distribution of the *template* to which the actual is *ultimately aligned*, rather than about the actual itself.⁸ The *alignment* of the actual to its ultimate template will not be changed, even if it is an array element or regular section; in fact, that is the essential meaning of the **INHERIT** directive. (E.g., if the actual is a regular section of an array, it will generally be aligned with a subset of the elements of some template—which is understood to be the array itself if the array is not explicitly aligned—and this alignment will be preserved.) However, that template may be redistributed (by a prescriptive distribution directive) or have its distribution asserted (by a descriptive distribution directive). Therefore, this combination of directives allows the alignment of an argument to its ultimate template be preserved, but the distribution of that template to be changed or asserted.⁹

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. E.g.:

```
!HPF$ DISTRIBUTE D * ONTO *
```

means that **D**’s distribution format, and the processors arrangement over which it is distributed, are inherited from the actual. (However if **INHERIT** is not specified, and the actual is a

⁸‘Ultimate alignment’ is explained at the end of section 2.2.

⁹Actually, it turns out that **DISTRIBUTE** can only be used in conjunction with **INHERIT** when the dummy has the same rank as the template with which the actual is ultimately aligned. E.g., it cannot be specified if the actual is an array element, or a section containing some scalar subscripts, unless the dimensions with scalar subscripts happen to be collapsed in the alignment of the actual argument array to its ultimate template. This limits the usefulness of this combination of directives! It is left as an exercise to the reader to work out why this limitation arises.

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*. Transcriptive and other forms can be mixed. E.g.:

```
!HPF$ DISTRIBUTE D * ONTO P
```

means the distribution format is inherited but the processors arrangement is prescribed (i.e. the actual may have been distributed over a different processors arrangement, in which case it will be redistributed over **P** using the same distribution format as before).

```
!HPF$ DISTRIBUTE D * ONTO *P
```

asserts that the actual is distributed over processors arrangement **P**, but its distribution format is inherited and could be anything.

```
!HPF$ DISTRIBUTE D (BLOCK) ONTO *
```

means that **D** is to be prescriptively block distributed onto whatever processors 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.

2.5.4 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, that is, of providing the caller with complete information about the procedure’s dummy arguments and, for a function, its dummy result (e.g. their types, shapes, whether they are used as input and/or output arguments, etc). The interface is automatically explicit for intrinsic and module procedures, and can be made explicit for external procedures by declaring an ‘**INTERFACE** block’ that contains the required information.

In HPF, if an explicit interface includes the mapping directives for the dummy arguments and result, the caller will remap the actual arguments 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.

2.6 The rôle of templates

A template is invaluable for providing a virtual array with which to align data arrays, when there is no suitable data array to serve that purpose.

For example, suppose that a regular section of an array is passed as an actual argument to a procedure, and it is desired to specify the mapping of the corresponding dummy argument such that it agrees exactly with that of the regular section, so that no data movement occurs. This requires an **ALIGN** directive to describe the alignment of the elements of the regular section. (Of course, one could always use an **INHERIT** directive to ensure that the regular section is received without being remapped, but this does not allow its mapping, and in particular its alignment, to be *described*.) However, it may well be that there is no suitable data array declared within the procedure to serve as the ‘align target’. (For example, a suitable align target might be an array of the same shape as that from which the regular

section was selected, as in the example below.) In that case, one could introduce a new array within the procedure to serve the required purpose, but that would waste storage space and obscure the program (and perhaps cause the compiler to generate warnings about variables that are declared but not used!) 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 object in the program. An example of this is shown in Figure 2.

```

      REAL a (100, 100)
!HPF PROCESSORS p (16, 16)
!HPF DISTRIBUTE a (BLOCK, BLOCK) ONTO p
      ...
      CALL sub (a (l1:u1, l2:u2), l1, u1, l2, u2)
      ...

      SUBROUTINE sub (d, l1, u1, l2, u2)
      REAL d (:, :)
      INTEGER l1, u1, l2, u2
!HPF TEMPLATE t (100, 100)
!HPF PROCESSORS p (16, 16)
!HPF ALIGN d (:, :) WITH *t (l1:u1, l2:u2)
!HPF DISTRIBUTE t (BLOCK, BLOCK) ONTO p

```

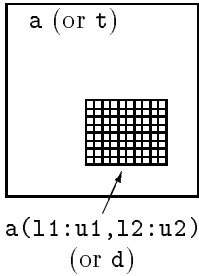


Figure 2: Describing a regular section using a template

In some applications, it may happen that there is no single data array that should totally cover the processor array. However, any object appearing in a **DISTRIBUTE** directive must totally cover the processor array—it is not possible to directly specify distribution over a subset of processors.¹⁰ Therefore, to specify the mapping of such objects they must first be aligned to an intermediary (i.e. a template) which is then distributed.

The above examples require the use of alignment to ‘embed’ the distribution of an array over a processor array. There are other cases where alignment must be used to describe a data mapping, e.g. it *must* be used to describe replication, or to permute dimensions in mapping a data array onto a processor array. In all of these case, templates can be used as the align target if no suitable data array is available.

There are also considerable stylistic advantages to using templates rather than arrays as alignment targets.

For example, if arrays are always aligned to 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). In contrast, when arrays are aligned to other arrays an arbitrarily complicated alignment tree can be constructed, which can make it difficult to identify the ‘root’ object with which a given array is ultimately aligned. To do so may involve scanning all of the **ALIGN** directives in a program unit to reconstruct the alignment tree, which is only known to be terminated when an align target appears in a **DISTRIBUTE** directive (and incidentally, it is not mandatory to explicitly distribute the root object, which complicates the investigation). Furthermore, it can be very difficult to work out the ultimate alignment

¹⁰ An exception would appear to occur if the distributee is a dummy argument that has appeared in an **INHERIT** directive—then the dummy argument may not totally cover the processor array if the corresponding actual argument is an array section. However, in that case the **DISTRIBUTE** directive actually refers to the *parent* array from which the section is selected, which *is* distributed over the whole processor array.

of arrays that are indirectly aligned to the root, and even more difficult to establish the relative alignments of arrays on different branches of the tree.

A commonly occurring situation is that several conforming arrays are to be related by an identity alignment (i.e. so that corresponding elements of all the arrays are aligned). In this case it is clearer to align them all to a single template, rather than to arbitrarily choose one as the ‘alignment root’ to which all the others are aligned, or to link them together in an ‘alignment chain’.

Finally, the root object of an alignment tree indicates the maximum 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.

2.7 Realignment and redistribution

The mapping directives that we have described so far are effectively declarations, and must appear among the declarations of a program unit.

There are also executable forms of the **ALIGN** and **DISTRIBUTE** directives, namely **REALIGN** and **REDISTRIBUTE**, which perform dynamic remapping of 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 dynamic remapping). If a dummy argument is dynamically realigned or redistributed, its original mapping is restored just before the procedure returns, so an actual argument cannot be remapped as a side-effect of a procedure call. 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 **SAVE**d variables cannot be dynamically remapped.

An object that has been aligned (i.e. has appeared as an alignee) 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. When such an object is redistributed, it ‘carries’ with it all the arrays aligned to it, so the alignment relations are preserved. Therefore, redistribution 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 to it). Interior nodes of an alignment tree *can* be realigned. 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 aligned to 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.

Any object (data object or template) that may be subject to realignment or redistribution must be specified in a **DYNAMIC** directive, e.g.:

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

2.8 Other issues

This description of HPF data mapping is by no means complete. For example, there are alternative syntactic forms for all of the directives, analogous to the new style of declarations in Fortran 90, which we have not mentioned. Nor have we mentioned the handling of allocatable arrays or pointers. For full details the reader is referred to [HPFF, 1993a].

3 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.

HPF adds to this a number of extra facilities for expressing data parallelism and concurrency, namely a **FORALL** statement and construct, **PURE** procedures, and an **INDEPENDENT** directive. We shall describe these features in this section.

FORALL has many properties in common with Fortran 90 array assignments, so we start with a brief review of the pertinent features of the latter.

3.1 Fortran 90 array assignments

An array assignment allows the simultaneous assignment to a whole array or a subset of its elements (an *array section*). For example, given arrays declared as:

```
REAL A(10), B(10), C(10)
```

the statement:

```
A = B + C
```

assigns to each element of **A** the sum of the corresponding elements of **B** and **C**, and the constituent elemental additions and assignments can be performed in parallel.

The semantics of an array assignment is that all elements of the expression on the right-hand side are fully evaluated, in any order and therefore potentially in parallel, and *then* the results are assigned to the corresponding elements of the assignment variable, again in any order and thus potentially in parallel. Therefore, if the evaluation of the right-hand side involves references to any part of the assignment variable, its *old* values prior to the assignment are used.¹¹ For example:

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

has the effect of setting each of the elements **A(2:9)** equal to the average of the old values its nearest neighbours. It is not the same as the apparently similar **DO**-loop:

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

In the latter, iterations $i > 2$ use the *new* value of **A(i-1)**, which was assigned in the previous iteration, together with the *old* value of **A(i+1)**, which has not been updated yet. Therefore this **DO**-loop contains dependences which fix the order of its iterations and thus force them to be executed sequentially. On the other hand, the evaluation of an array expression is inherently data-parallel¹², regardless of its context.

Another property of Fortran 90 array assignments is that they are designed to be *deterministic*, that is, the value assigned to each element of the assignment variable is well-defined even though the order of evaluation of the elemental expressions, and the order of the elemental assignments, are undefined. Two rules in particular are imposed to ensure this:

¹¹Of course, if the assigned array section is *not* referenced during the evaluation of the right-hand side expression, the synchronisation point between the expression evaluation and assignment, which is implicit in the semantics, is unnecessary and can be removed.

¹²except for function references and scalar subexpressions within the expression, which may or may not admit data-parallelism.

- (i) if the assignment variable is an *irregular section*¹³, all of its elements must be distinct (otherwise an element could be assigned multiple values and its final value would depend on the order of assignment);
- (ii) a function evaluation in an assignment statement must not have a side-effect that would affect the evaluation of any other entity in the statement.

It is possible to *mask* an array assignment, so that the expression evaluation and assignment are only performed for certain elements of the array variable, using a **WHERE** statement. E.g.:

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

evaluates the *mask* array expression $(A(:) > 0.0)$, and then evaluates the right-hand side and performs the assignment only for elements corresponding to elements of the mask that are **.TRUE.** (i.e. elements where $A(i) > 0.0$).¹⁴ A **WHERE-ELSEWHERE** construct is also provided, so that a single mask expression can mask a sequence of array assignments.

3.2 FORALL statement and construct

HPF extends Fortran 90's facilities for expressing data parallel assignment by introducing a **FORALL** statement and construct.

The **FORALL** statement allows a data parallel assignment to a group of array elements to be expressed in terms of its constituent elemental assignments. E.g. with arrays **A**, **B** and **C** declared as in the last section:

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

means the same thing as the array assignment $A = B + C$.

It will be helpful to introduce some terminology for the parts of a **FORALL** statement. In the above example, the index **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 similar to a **DO**-loop over array element assignments (or at least, a **FORALL** construct looks like that!). However, its semantics are the same as those of 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))
```

is equivalent to:

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

¹³ An *irregular section* is formed using one or more *vector subscripts*, whose elements provide the subscript values for that dimension of the array section. E.g. if **V** is an integer vector of length 3 whose elements have the values (5,1,2), then **A(V)** is a section comprising the elements **(A(5),A(1),A(2))**.

¹⁴ However, non-elemental function references and subscript expressions within the assignment statement are fully evaluated regardless of the mask.

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 paper we occasionally 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 this is not in standard usage—currently there does not appear to be a recommended or commonly accepted term for this purpose.

The **FORALL** header can declare multiple indices, and their ranges may optionally specify a stride. The general form for specifying the range of a **FORALL** index is similar to Fortran 90’s subscript triplet notation, except that the lower and upper bounds *must* be specified, i.e. it 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. Thus an assignment to a multi-dimensional regular section can be expressed straightforwardly, e.g.:

```
FORALL (i=2:10:2, j=1:m:n) A(i,j) = B(i,j)
```

means the same thing as:

```
A (2:10:2, 1:m:n) = B (2:10:2, 1:m:n)
```

The bounds and stride of a **FORALL** index must not depend on **FORALL** indices, so they can only describe rectangular sections directly (but see later).

A **FORALL** assignment need not be scalar—it can be an array assignment. E.g., the last example could also be written as:

```
FORALL (i=2:10:2) A(i, 1:m:n) = B(i, 1:m:n)
```

Also, subscripts in the **FORALL** assignment can be general expressions—they do not have to be simply the **FORALL** indices. The only condition is that a **FORALL** statement must not assign multiple values to any element, which would be non-deterministic given that the order of the individual assignments is undefined. This is analogous to the Fortran 90 rule that, if an irregular section is assigned, all of its elements must be distinct. Thus:

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

is legal only if **indx** contains no repeated values. Incidentally, a consequence of this condition is every **FORALL** index must appear among subscripts of the **FORALL** assignment variable. E.g.:

```
FORALL (i=1:10, j=1:5) A(i) = B(i,j)
```

is illegal, as each **A(i)** is assigned 5 values because of the index **j**. However, the following *is* legal:

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

(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 contain a scalar mask (i.e. logical) expression, in which case the **FORALL** assignment (including the evaluation of its right-hand side) is only executed for those 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 the **FORALL** statement duplicates the functionality already provided by Fortran 90's array syntax. However, the **FORALL** statement is often clearer and more concise, and is certainly more flexible, 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 (e.g. elemental assignments in **DO**-loops) in these cases, which hide the data-parallelism and require 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:

- When dimensional permutation is involved. E.g.:

```
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/))
```

- To avoid the conformance rules for array assignments. E.g:

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

versus:

```
A = SPREAD (B, DIM = 2, NCOPIES = n)
```

- To express subscript-dependent values. E.g., the following initialises 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, and sometimes impossible, to express in array syntax. E.g. the Fortran 90 equivalent of the above is:

```
even = (MOD (SPREAD ((/i,i=1,m/), DIM = 2, NCOPIES = n) +  
              SPREAD ((/j,j=1,n/), DIM = 1, NCOPIES = m), 2) == 0)
```

- To express non-rectangular array sections. E.g.:

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

- To express more general array access patterns. E.g.:

```
FORALL (i=1:n) ... A(i,jvec(i)) ...
```

accesses one element from each row of the array, the particular element selected from each row being determined by the values in vector **jvec**.

In fact, it is possible to select elements from an array in any fashion to form another array of any shape. E.g., the following copies elements from a 2-dimensional to a 3-dimensional array; the shapes of the arrays are totally independent, and each element of the destination array can correspond to any element of the source array.

```
FORALL (i=1:L, j=1:M, k=1:N) B(i,j,k) = A(ivec(i,j,k), jvec(i,j,k))
```

This is a much more general type of ‘irregular section’ than can be expressed using Fortran 90 vector subscript notation. The latter can only be used to form a 1 or 2 dimensional section from **A**. If **B** were 2-dimensional in this example, omitting index **k**, it could still only be expressed as an array assignment if **ivec** were independent of **j** and **jvec** independent of **i**.

The next example simulates 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 directly an array assignment, however, as this pattern of row and column shifts cannot be expressed by array sections. For clarity, all of the subscript ranges are assumed to be 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 1 each time, and the results are accumulated into *C*, the matrix product $C = AB$ will be produced.

- Finally, a **FORALL** statement must be used when the constituent elemental assignment involves a reference to a non-elemental function. E.g., the following is a completely data parallel expression of matrix multiplication, which takes the product of matrices $A(m,k)$ and $B(k,n)$ and assigns the result to matrix $C(m,n)$:

```
FORALL (i=1:m, j=1:n) C(i,j) = DOTPRODUCT (A(i,:) * B(:,j))
```

This cannot be written as an array assignment to the whole of **C** because of the reference to the non-elemental intrinsic function **DOTPRODUCT**. 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. 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.

For completeness, we mention that an assignment in a **FORALL** statement or construct may be a pointer assignment rather than a normal assignment (although this is likely to be of limited use!).

Figure 3 illustrates the use of **FORALL** in a code for performing the LU decomposition of a square matrix **A**. This is the core of an algorithm for solving a set of linear equations $\underline{AX} = \underline{B}$. For brevity this example omits ‘pivotting’, which would normally be used to

```

SUBROUTINE LU_DECOMP (A, n)
  INTEGER i, n, r, r1
  REAL A (n,n)
!HPF$ DISTRIBUTE A (CYCLIC, CYCLIC)
!-----!
! LU decomposition of matrix 'A'. The result overwrites 'A' !
!-----!

  DO r = 1,n
    r1 = r+1
    A(r,r1:n) = A(r,r1:n) / A(r,r)
    FORALL (i = r1:n) A(i,r1:n) = A(i,r1:n) - A(i,r) * A(r,r1:n)
  END DO
END SUBROUTINE

```

Figure 3: LU decomposition (without pivoting)

improve the numerical stability of the algorithm. The array **A** is distributed over a 2-dimensional processor array.

Roughly speaking, in **DO**-loop iteration r , the **FORALL** statement subtracts row r of **A** from all of the rows $(r + 1)$ to n in parallel. (Speaking more precisely, row r is scaled by $A(i,r)/A(r,r)$ before being subtracted from each row $i \in [r + 1 : n]$, and only the sections $[r + 1 : n]$ of the rows are operated upon.)

HPF does not specify an implementation strategy for **FORALL**, but for a distributed memory machine one could imagine that a good implementation might broadcast the section $A(r, r+1:n)$ of row r so that it is aligned with the sections $A(i, r+1:n)$ of all rows $i \in [r + 1 : n]$, and then perform the operation on these rows in parallel (at least to the extent allowed by the number of processors used to store the rows).

Incidentally, notice that array **A** has been distributed cyclically in both dimensions. The section of **A** that is involved in the computation diminishes as the execution progresses (i.e. iteration r only involves the section $A(r:n, r:n)$). Therefore, if **A** were distributed blockwise over the 2-dimensional processor array, the ‘area’ of the processor array that is utilised would diminish correspondingly. Assuming that **A** is larger than the processor array, giving it a cyclic distribution helps to spread-out the workload.

3.3 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 *non-deterministic* (e.g. if one function invocation writes to a variable that another reads, or two invocations write different values to the same variable, then the overall behaviour 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. If an ordinary function is referenced concurrently, i.e. is executed on a subset of the processors that are allocated to the program, it might access variables stored in the local memories of processors that it is not executing on. This cannot be supported on distributed memory architectures using pure message-passing, as the latter requires that the processors at both ends of a communication execute the communication instruction. To support this behaviour 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 a **FORALL** scalar mask expression.¹⁵

However, HPF defines a special class of side-effect free functions called *pure functions*, which can be used in these contexts. They are denoted by adding the keyword **PURE** before **FUNCTION** keyword in the function header statement, and they must satisfy a number of constraints, which are checkable at compile-time, to ensure that they are both safe (i.e. side-effect free) and efficiently implementable under concurrent reference. In outline, the constraints are as follows:

- To ensure side-effect freedom, pure functions must *not*:
 - modify dummy arguments or global variables (e.g. they cannot be assigned or pointer-assigned, or passed as actual arguments that may be modified);
 - **SAVE** local variables (which would be a side-effect between different executions of the same function);
 - reference any non-pure procedures;
 - perform external I/O;
 - contain **PAUSE** or **STOP** statements;
 - perform dynamic realignment or redistribution;
- To enable concurrent references of pure functions to be efficiently implemented, there are restrictions on data mapping (which are justified later). They are:
 - the dummy arguments and result can only be aligned among themselves;
 - local variables can only be aligned among themselves or with dummy arguments or the dummy result.

Apart from this, the local variables, dummy arguments and dummy result may not be subject to any other type of mapping directives. The mapping of global variables is not constrained however.

‘Pure’ subroutines may also be defined, and must satisfy the same constraints as pure functions except that they may modify their dummy arguments. They are useful for a variety of purposes, e.g. so that subroutines can be called from within pure functions and subroutines, 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 in any way that a normal procedure can. However, a procedure is *required* to be pure if it is used in any of the following contexts:

¹⁵However, normal functions *can* be referenced in the bound and stride expressions that define **FORALL** indices, as these are only evaluated once, not multiple times.

- in a **FORALL** assignment or mask expression, or a statement in a **FORALL** construct;
- within the body of a pure procedure;
- as an actual argument in a pure procedure reference.

When a procedure is used in a context that requires it to be pure, its interface must be explicit, and both its interface and definition must specify the **PURE** keyword and also the **INTENT**¹⁶ of all dummy arguments except for pointer and procedure arguments (though admittedly this is redundant for a pure function as all of 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. A statement function is pure if and only if all functions that it references are pure. Of the intrinsic subroutines, only **MVBITS** is pure; the others are not pure as they perform I/O.

The restrictions on data mapping in pure procedures may warrant some explanation. The reason for them is that the procedure may be invoked concurrently, with each invocation active on a subset of processors specific to that invocation. As we have explained, on multiprocessor systems without shared memory support, the data accessed by a procedure must be mapped to the local memories of the processors that are executing it. For efficiency, the caller should have the freedom to choose according to context the processor subset on which to execute any particular pure procedure reference, e.g. so as to maximise concurrency in a **FORALL**, and/or reduce communication, taking into account the mappings of other terms in an expression, the assignment variable, etc. This implies that, on platforms without shared memory support, it must also have the freedom to map the procedure's actual arguments, result and local variables to the chosen processor subset (which, incidentally, may involve dynamically remapping the actual arguments), just as it has this freedom generally for variables in an expression. Therefore, a dummy argument or result may not appear in a mapping directive that fixes its location with respect to the processor array (e.g. it may not be aligned with a global variable or template, or be explicitly distributed, or even appear in an **INHERIT** directive, all of which would remove the caller's freedom to determine the actual's mapping as described above). The only type of mapping information that may be specified for the dummy arguments and result is their alignment with each other; this will provide useful information to the caller about their required *relative* mappings. For similar reasons, local variables may be aligned with the dummy arguments or result (either directly or through other local variables), but may not have arbitrary mappings.¹⁷

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 that the implementation can remap the actual arguments as it sees fit (hopefully in an efficient way). This is one place where the programmer is to a large extent relieved of the burden of worrying about data mapping (expressions being another).

These considerations can be illustrated by considering again the LU decomposition example of Figure 3. One could rewrite the **FORALL** statement using a pure function to do the row subtraction operation (which is obviously not worthwhile in this case, but serves as an illustration):

¹⁶Dummy arguments can be specified as **INTENT(IN)**, **(OUT)** or **(INOUT)**, meaning respectively that they are read, written, or both.

¹⁷However, the implementation on non shared-memory platforms is still complicated by the fact that pure procedures can access global variables whose mapping *is* fixed with respect to the processor array.

```

    PURE FUNCTION row_op (r1, factor, r2)
      REAL, INTENT (IN) :: r1 (:), factor, r2 (:)
      REAL               :: row_op (SIZE(r1))
!HPF$ ALIGN  r2(:)      WITH  r1(:)
!HPF$ ALIGN  row_op(:)  WITH  r1(:)

      row_op = r1 - factor * r2
    END FUNCTION

    ...
    FORALL (i=r1:n) A(i,r1:n) = row_op (A(i,r1:n), A(i,r), A(r,r1:n))

```

The **FORALL** statement could be implemented exactly as before: the implementation could broadcast row **A(r, r1:n)** to be aligned with each of the rows **A(i, r1:n)** ($i \in [r+1 : n]$), according to the alignment specified within the pure function,¹⁸ and then invoke the pure function in parallel over these rows. This implementation might easily be ruled out if the programmer could specify arbitrary data mapping directives for **row_op**'s dummy arguments and local variables.

Another point to note is that, except for the prohibition of **PAUSE** and **STOP** statements, pure functions have no constraints on their internal control flow. In the context of concurrent reference within a **FORALL**, this allows some degree of ‘MIMD’ or ‘functional’ parallelism in an HPF program, as different concurrent invocations of the function can execute different code. This is illustrated by the example in Figure 4, which plots the Mandelbröt set over a square lattice of points by invoking the pure function **mandel** at each grid point. The number of iterations executed by **mandel**, and the branch it takes in the **IF** construct, will differ for different invocations. Apart from pure procedure references in **FORALL**, MIMD parallelism can also arise via ‘independent’ **DO**-loops and ‘extrinsic’ procedure references (both of which are briefly introduced later).

Finally, we mention that an early draft of HPF allowed pure procedures with scalar dummy arguments and result to be referenced *elementally*. This means that the procedure could be invoked with conforming array-valued input arguments, and would return a conforming array result (for a function) or output arguments (for a subroutine), each element of which has the same value as if the procedure were applied to the corresponding elements of the input arguments. Many of Fortran 90's intrinsic functions are classified as ‘elemental’ and can be referenced in this way. E.g., the scalar intrinsic function **sin(x)** can be invoked with an array argument:

```
a(1:10) = sin (b(1:10))
```

In the Mandelbröt example, elemental reference would allow the **FORALL** statement to be replaced by:

```
n = mandel (coord)
```

where **coord** is a complex array whose elements represent grid point coordinates. This facility has been dropped from HPF merely to reduce the number of extensions to Fortran 90, and is presented in the HPF ‘Journal of Development’ [HPFF, 1993b].

¹⁸ We have said that a pure function referenced in a **FORALL** must have an explicit interface, so the caller is aware of the data mapping directives within the function.


```

PURE INTEGER FUNCTION mandel (c)
  COMPLEX, INTENT (IN) :: c
  COMPLEX :: z
  INTEGER :: itn
!-----!
! 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 of -1 is returned.                        !
!-----!
  z = c
  itn = 0
  DO WHILE (ABS (z) < 2.0 .AND. itn < 100)
    z = z*z + c
    itn = itn + 1
  ENDDO
  IF (ABS (z) < 2.0) THEN
    mandel = -1
  ELSE
    mandel = itn
  ENDIF
END FUNCTION mandel

...
REAL n (-100:100, -100:100) ! #itns to diverge to |z| >= 2
!HPF$ DISTRIBUTE n_itns (BLOCK, BLOCK)
REAL, PARAMETER :: dx = 0.02, dy = 0.02
...
FORALL (i= -100:100, j= -100:100) n(i,j) = mandel (CMPLX(i*dx, j*dy))

```

Figure 4: Using a pure function to plot the Mandelbröt set.

3.4 INDEPENDENT directive

HPF also introduces an **INDEPENDENT** directive, which can precede a **DO** loop or **FORALL** statement or construct.

If it precedes a **DO**-loop it asserts that the loop iterations are ‘independent’ and so can be executed in any order and therefore potentially concurrently. This requires that several conditions are satisfied, namely: that variables that are written in one iteration are not referenced (i.e. read or written) in any other iteration; that no iteration transfers control out of the loop or executes a **PAUSE**, **STOP** or **EXIT** statement; that no two iterations perform any external I/O operations (except **INQUIRE**) to the same ‘I/O unit’; and that no iteration performs dynamic remapping of data that are referenced or remapped by any other iteration. These are assertions about *behaviour*, and do not imply any syntactic constraints; e.g. all the iterations of an independent **DO**-loop may contain an assignment to the same variable, provided not more than one iteration actually *performs* the assignment. There are no restrictions on control flow, procedure calls, etc, within the loop.

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, in violation of the conditions). This **DO**-loop is therefore equivalent to the Fortran 90 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 of the iteration. This clause also constitutes an assertion that this modification does not change the meaning of the program, i.e. that 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 is really derived from Fortran dialects for such machines). This is because of the complete generality of data references allowed within them (which may inhibit concurrent execution on distributed memory machines using pure message-passing) and of control flow (which is likely to prevent concurrent execution on SIMD machines). Therefore, if a program is intended to be run on non shared-memory architectures, we recommend that the programmer uses array or **FORALL** syntax rather than independent **DO**-loops wherever possible in order to maximise the performance of the program.

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. (The other conditions listed above do not apply here, as they refer to behaviour that is not possible in a **FORALL**.) E.g.:

```
!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. $m \leq n$).

FORALL already has parallel semantics. The condition asserted by **INDEPENDENT** simply removes the various synchronisation points between the instances of a **FORALL** that are implicit in its semantics, i.e. between evaluating the right-hand side expressions and performing the assignments of an assignment statement, and between successive statements in a **FORALL** construct. In particular this means that assignments in **FORALL** can proceed directly rather than through temporary intermediate storage, which is a useful optimisation.

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

4 Other HPF extensions

HPF includes a number of other extensions which we summarise here. We do not describe them in detail due to lack of time and space, not because they are unimportant. The reader is referred to [HPFF, 1993a] for full details.

The remaining HPF extensions are as follows:

- New intrinsic functions, namely:
 - functions for system enquiry: **NUMBER_OF_PROCESSORS** and **PROCESSOR_SHAPE**, which return information about the *physical* processors on which the program is running (as distinct from the abstract processors that are declared and used in data mapping directives);
 - a computational intrinsic function **ILEN**;
- and an extension to a couple of the Fortran 90 intrinsic functions, namely:
 - **MINLOC** and **MAXLOC** are given an extra optional argument **DIM** for finding the locations of the maximum and minimum elements along a given dimension.
- A *standard library* of procedures in a module called **HPF_LIBRARY**, containing:
 - subroutines for enquiring about data mapping: **HPF_ALIGNMENT**, **HPF_TEMPLATE** and **HPF_DISTRIBUTION**;
 - bit manipulation functions: **LEADZ**, **POPCNT** and **POPPAR**;
 - new array reduction functions: **IALL**, **IANY**, **IPARITY** and **PARITY**, which are analogous to the Fortran 90 intrinsic reduction functions **ALL** and **ANY**, but apply the operators **IAND** (bitwise AND), **IOR** (bitwise OR), **IEOR** (bitwise EOR) and **.NEQV.** (logical EOR) respectively;
 - more general reduction functions: **XXX_SCATTER** for ‘combining scatter’, and **XXX_PREFIX** and **XXX_SUFFIX** for ‘parallel prefix’ and ‘suffix’, where **XXX** is any of the available reduction operations;
 - array sorting functions: **GRADE_UP** and **GRADE_DOWN**.

- An *extrinsic interface*. This provides an escape mechanism from HPF to another programming model and/or language, which is achieved by calling non-HPF procedures called *extrinsic procedures*. Their interface must be explicit (the so-called ‘extrinsic interface’), 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, to hand-tune critical kernels, etc, at the expense of non-portability.

In fact, HPF defines one particular type of extrinsic model, called ‘**HPF_LOCAL**’, which 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.

- A **SEQUENCE** directive, which *must* be used to identify all variables and common blocks are subject to sequence and/or storage association. These associations imply restrictions on their mapping.

5 Subset HPF

An official subset of HPF, called ‘Subset HPF’, has been defined to facilitate rapid initial implementation. It is intended that it will provide a portable interim HPF capability at an early stage, while the implementation of the full language will take longer.

It is based on a subset of Fortran 90, to avoid delaying initial implementations by the need to base them on full Fortran 90, compilers for which are still not widely available. The Fortran 90 subset includes:

- all of Fortran 77;
- all of the non-character array features and most of the intrinsics of Fortran 90;
- dynamic memory allocation;
- interface blocks without generic specifications or module procedure statements;
- optional and keyword arguments;
- new-style type declarations;
- various lexical and syntactic improvements.

Put another way, roughly speaking Subset HPF includes all of Fortran 90 *except for* pointers, derived data types, defined operations and assignments, modules, internal procedures, generic interfaces, **CASE** and ‘**DO forever**’ constructs, **EXIT**, **CYCLE** and **NAMelist** statements, **KIND** parameters and free source form. It also has some restrictions on the new intrinsic procedures. These lists are merely indicative, and the reader is referred to [HPFF, 1993a] for precise details.

Subset HPF is also restricted in terms of the HPF extensions. It *excludes*:

- dynamic data redistribution;

- the **INHERIT** directive unless is accompanied by a prescriptive or descriptive distribution directive;
- the **FORALL** construct (though the **FORALL** *statement* is included);
- **PURE** procedures;
- the **EXTRINSIC** interface;
- the HPF standard library (though the HPF *intrinsic*s are included).

6 Conclusions

We believe that HPF is a significant step forward in simplifying the programming of data parallel applications on parallel computers, particularly distributed-memory MIMD systems. It is based on a simple and familiar programming language and offers wide portability, a simple migration method for existing Fortran codes, and the promise of high performance, so there is every chance that it will rapidly become one of the foremost languages for scientific and engineering applications on such platforms. Indeed, the lack until now of a programming language with these desirable features for distributed-memory MIMD architectures has been the main reason for their low user acceptance, despite their potentially high performance and cost effectiveness.

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. The programmer will need a good understanding of the program and of the HPF execution model in order to map data effectively. In addition, we anticipate that some old ‘dusty deck’ Fortran programs may need to be significantly re-written 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 ‘optimisations’ are ‘clean’, in that they should greatly improve code legibility as well as efficiency.

Acknowledgements

The author would like to thank Bryan Carpenter for providing the LU decomposition example and for useful discussions.

References

- [HPFF, 1993a] High Performance Fortran Forum (1993) High Performance Fortran Language Specification. Technical report CRPC-TR-92225, Center for Research in Parallel Computation, Rice University, Houston, TX. To appear in *Sci. Prog.*, **2** (1), July 1993. Also available by anonymous FTP from Rice University, from **titan.cs.rice.edu** in directory **/public/HPFF/draft**.
- [HPFF, 1993b] High Performance Fortran Forum (1993) High Performance Fortran Journal of Development. Available by anonymous FTP as above.