High Performance Fortran 2.0

John Merlin and Barbara Chapman,

VCPC (European Center for Parallel Computing at Vienna),
University of Vienna,
Liechtensteinstrasse 22,
A-1090 Vienna, Austria.

jhm@vcpc.univie.ac.at

November 24, 1997

Abstract

High Performance Fortran (HPF) is an informal standard for extensions to Fortran 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, extra facilities for expressing data parallel and concurrent execution, and a mechanism for interfacing HPF to other languages and programming models.

This paper provides a comprehensive tutorial introduction to HPF 2.0, the latest version of the HPF standard which was published in early 1997. It also outlines the history of the HPF language development and lists some World Wide Web sites where information about HPF compilers and tools, tutorials, codes and projects can be found.

1 Introduction

High Performance Fortran (HPF) is an informal language standard which aims to 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 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 removes this burden from the programmer. It comprises a set of extensions to standard Fortran. 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, for example 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. 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.

This paper provides a comprehensive introduction to HPF 2.0, the latest version of the HPF standard which was published in early 1997. Section 2 outlines the history of the HPF language development. Sections 3–6 explain HPF's data distribution extensions, namely the `DISTRIBUTE` and `ALIGN` directives, the distribution of procedure arguments, and the `TEMPLATE` directive. Section 7 describes the restrictions imposed by HPF on the use of sequence and storage association for distributed variables. Section 8 presents HPF's extensions for expressing data parallel and concurrent execution. Section 9 describes the extrinsic mechanism, whereby HPF can call non-HPF procedures. Sections 10 and 11 summarise the HPF intrinsic functions and standard library, and the HPF 2.0 'Approved Extensions'. Finally Section 12 provides a survey of some World Wide Web sites where information can be found about HPF compilers and tools, tutorials, example codes and projects.

2 History of HPF

The HPF extensions were developed 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 academic research groups in the field of parallel computation. There have been three rounds of HPFF meetings so far.

2.1 HPF 1.0

The first version of the language, HPF 1.0, was developed between March 1992 and May 1993, following an initial kick-off meeting convened in January 1992 by Ken Kennedy of Rice University and Geoffrey Fox of Syracuse University. The design of HPF's data distribution features was strongly influenced by Fortran D and Fortran 90D [1, 2] and Vienna Fortran [3]. Significant inputs to the language development were also provided by a number of other research prototype parallelisation systems based on language extensions for specifying data distribution, such as Distributed Fortran 90 [4, 5] and Pandore C [6], as well as by Fortran dialects and proposals from vendors such as Digital [7], Convex, Cray [8], IBM [9], Maspar [10] and Thinking Machines [11], together with inputs from a variety of other sources; see [12] for further references.

The HPF 1.0 language definition was published in [12]. This document also defined an official subset of the language, 'Subset HPF', to facilitate early implementation. In practice, all early HPF implementations, which started to appear in about 1994, concentrated on the Subset HPF features rather than attempting to support all of HPF.

A number of features that were considered but not accepted into HPF 1.0 were presented in a separate document, the 'HPF Journal of Development' [13]. 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 so were documented so that they could provide input to future language design activities.

A textbook about HPF-1 has been published [14], as have a number of HPF-1 language tutorials, e.g. [15].

2.2 HPF 1.1

A second set of meetings from April 1994 to October 1994 concentrated on corrections, clarifications and interpretations of HPF 1.0, resulting in the production of a revised and corrected version of the language specification, HPF 1.1 [16]. Some requirements for HPF 2.0 were also identified at that time [17].

2.3 HPF 2.0

A third set of meetings was held from January 1995 to December 1996 to develop further extensions to HPF. Their aim was to broaden HPF's applicability by providing features such as enhanced data distributions, task parallelism and computation control, parallel I/O, and directives to assist communication optimisation [17].

However, it became clear that vendors were reluctant to greatly extend the basic HPF language for fear of delaying commercial implementations and/or encouraging partial implementations, thus undermining HPF's credibility and use. The outcome was to define a new HPF base standard, HPF 2.0, which in terms of its extensions is quite similar to Subset HPF-1. In fact the main difference between Subset HPF-1 and HPF 2.0 is that the former was based on Fortran 77 plus a subset of Fortran 90 features that were considered important for parallelism, for example array syntax, while HPF 2.0 is based on full Fortran 95[1] [18].

All other HPF extensions, both new and old, are designated 'HPF 2.0 Approved Extensions'. The idea is that a standard-conforming HPF-2 compiler must provide full support for the HPF 2.0 features, but is not required to support any of the Approved Extensions. Presumably the Approved Extensions will tend to provided only if there is sufficient demand from users. Also the 'Approved Extension' status should allow more flexibility to deviate from the detailed specification, possibly allowing improvements in the light of implementation and user experience. Those features that turn out to be widely used can then be incorporated in the next revision of the HPF base standard. The specification of the HPF 2.0 language and Approved Extensions can be found in [19].

This paper provides an informal introduction to HPF 2.0, assuming familiarity with Fortran 90. We shall concentrate on the HPF 2.0 base language, except for section 11 which summarises the Approved Extensions. We start with HPF's most fundamental and important extensions, namely its extensions for specifying data distribution, which are the subject of the next few sections.

3 Data distribution

'*Data mapping*' is the HPF term for the allocation of data to multiple memories. HPF's data mapping extensions are its most fundamental features.

Data mapping is specified in HPF by '*directives*'. These are structured Fortran comments that are distinguished by starting with the characters 'HPF$' immediately after the comment

[1] Fortran 95 is the latest revision of Fortran, which is expected to be published in late 1997 or early 1998. Its main enhancements compared with Fortran 90 are features from HPF-1, namely FORALL, PURE procedures, and the ability to elementally reference PURE procedures (the latter of which was proposed in the HPF Journal of Development although not actually a part of HPF-1). Thus one could say that HPF-2 is based on Fortran 95, and Fortran 95 is based on HPF-1!

character, that is, 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, so an HPF program can even be compiled by a normal Fortran compiler for a single processor machine. This is acceptable as they do not affect the *semantics* of a program, that is, they do not change its computations or results, except for possibly affecting the order of computations when this is not defined by the language, for instance the order of the operations in an intrinsic reduction function like SUM. The data mapping directives only affect a program's *performance*, not its *meaning*.

3.1 The PROCESSORS directive

The PROCESSORS directive declares and names one or more abstract processor arrangements, where a 'processor arrangement' means a processor array or a scalar, i.e. single, processor. For example:

```
!HPF$ PROCESSORS  p (4),  q (8, NUMBER_OF_PROCESSORS()/8), r
```

declares abstract processor arrays p and q of the given dimensions, and a single abstract processor r. Note that these are *abstract* processors. This allows implementations the freedom to abstract the processors declared in HPF from the real physical processors. For example the former may actually be '*processes*', and an implementation may be able to execute multiple processes concurrently on each physical processor.

Incidentally, 'NUMBER_OF_PROCESSORS()' is an HPF intrinsic function that returns the number of processors, or in some implementations processes, on which the program is executing (see Section 10.1).

Abstract processor arrays with different shapes and total sizes 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, that is, 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, processors arrangements must be declared locally in every program unit in which they are used.

3.2 The DISTRIBUTE directive

The DISTRIBUTE directive specifies how a data object is to be distributed over an abstract processor arrangement.

The distribution that can be specified for a scalar object is somewhat limited: it can only be distributed onto a scalar processor arrangement, i.e a single processor.[2] Therefore we shall concentrate on distributing arrays in the rest of this section.

In that case, a so-called *distribution format* is specified for each dimension of the *distributee*, i.e. the object that is distributed, which can be either:

[2] We will show later, in Section 4.3, how to store a copy of a scalar on every processor in a processor array, i.e. how to *replicate* it.

BLOCK [(*blocksize*)]

CYCLIC [(*blocksize*)], or

*

where [...] encloses an optional item. For simplicity we shall illustrate them for a 1-dimensional array:

```
REAL  a (12)
```

distributed over a 1-dimensional processor array:

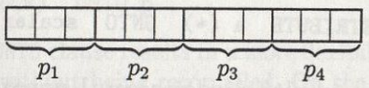
```
!HPF$ PROCESSORS  p (4)
```

BLOCK means that the elements are divided into equal, or nearly equal, size 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 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:

```
!HPF$ DISTRIBUTE  a (BLOCK)  ONTO  p
```

results in the elements being allocated to processors as follows:



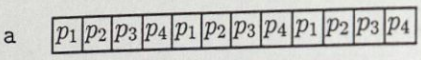
An explicit blocksize can be specified in parentheses after the BLOCK keyword. (By definition, BLOCK means BLOCK($\lceil n/p \rceil$) as we have said.) The specified blocksize must be such that the elements do not 'wrap around' the processor array. Thus in this example the blocksize must be ≥ 3, as a blocksize of 1 or 2 would cause 'wrap-around' and be erroneous. To allow 'wrap around' a CYCLIC 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 utilisation (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.

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.

For example:

```
!HPF$ DISTRIBUTE  a (CYCLIC)  ONTO  p
```

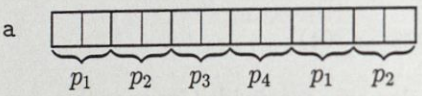
results in the following allocation of elements to processors:



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  a (CYCLIC (2))  ONTO  p
```

results in the following distribution of elements to processors:



Note that BLOCK(b) and CYCLIC(b) are identical in the case when there is no wrap around, i.e. when $n \leq bp$. 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 given shortly.

* means that the corresponding distributee dimension is *collapsed*, that is, *not* distributed. If this is applied to our one-dimensional array a, the target processor arrangement must be scalar:

```
!HPF$ DISTRIBUTE  a (*)  ONTO  scalar_proc
```

3.3 Distributing multi-dimensional arrays

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'.

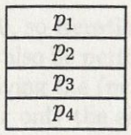
A distribution format must be specified for every dimension of a distributee. Each dimension is either distributed over a processor array dimension or is collapsed. Therefore, the number of BLOCK and CYCLIC entries must equal the number of processor array dimensions, and the nth distributee dimension with such an entry is distributed over the nth processor array dimension. For example:

```
      REAL  b (10,10,10,10)
!HPF$ PROCESSORS  q (4,4)
!HPF$ DISTRIBUTE  b (BLOCK, *, CYCLIC(2), *)  ONTO  q
```

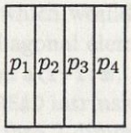
means that b's first dimension is distributed BLOCK over the first dimension of q, its third dimension is distributed CYCLIC(2) over the second dimension of q, and dimensions 2 and 4 of b are collapsed, that is, for fixed subscripts in the other dimensions, all subscript values in dimensions 2 and 4 are mapped to the same processor.

Here are some examples of distributing a 2-dimensional array c onto processor arrays p(4) and q(2,2):

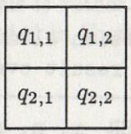
```
!HPF$ DISTRIBUTE  c (BLOCK, *)  ONTO  p
```




```
!HPF$ DISTRIBUTE  c (*, BLOCK)  ONTO  p
```




```
!HPF$ DISTRIBUTE  c (BLOCK, BLOCK)  ONTO  q
```



3.4 Omitting the ONTO clause

The ONTO clause may be omitted from a DISTRIBUTE directive, in which case the distribution is onto an implementation-dependent default processors arrangement. Typically the default processor array has a size equal to the number of processors the program is executed on. Thus:

```
!HPF$ DISTRIBUTE  a (BLOCK)
```

is usually equivalent to:

```
!HPF$ PROCESSORS  p (NUMBER_OF_PROCESSORS ())
!HPF$ DISTRIBUTE  a (BLOCK)  ONTO p
```

This means that omitting the ONTO clause results in a more flexibile program, as it can be run on any number of processors without being recompiled. On the other hand, specifying a constant processor array size should allow the compiler to generate faster code. Also, one cannot predict the shape of a default processor array with 2 or more dimensions, as it is undefined by the language and varies between HPF implementations. For example, the default 2-dimensional processor array could have shape `(1, NUMBER_OF_PROCESSORS())`, or `(NUMBER_OF_PROCESSORS(), 1)`, or anything in-between.

3.5 An example code—Gaussian elimination

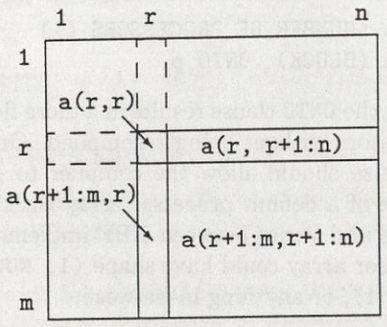
The example code that we shall use to illustrate HPF data mapping is the forward elimination phase of Gaussian elimination, shown in Figure 1. We use Fortran 90 array syntax rather than DO-loops where possible, as array syntax explicitly expresses the potential for data parallel execution and so is recommended (by us at least) for use with HPF. 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 iteration r, row r of A is divided by $A(r, r)$, and then each row $i > r$ has $(A(i, r) \times$ row $r)$ subtracted from it (potentially in parallel over the rows i).

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

END DO
```

(a): Code



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

Figure 1: Forward elimination phase of Gaussian elimination (without pivoting)

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 iteration r only the sections $[r + 1{:}n]$ of the rows are operated upon, as by definition the other elements become 0, except 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 non-zero.

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 2-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 (now included in Fortran 95) which allows it to be expressed more concisely, as we shall see later.

3.5.1 Distributing the data

Notice that the section of `a` that is involved in the computation diminishes as the execution progresses, that is, iteration r only involves the section `a(r:m, r:n)`—see Figure 1. Therefore, if `a` were block distributed, the 'area' of the processor array that is utilised would diminish correspondingly. This example is therefore a candidate for cyclic distribution, e.g.:

```
!HPF$ DISTRIBUTE  a (*, CYCLIC)     ! ...or (CYCLIC,*)
```

onto a 1-dimensional processor array, or perhaps

```
!HPF$ PROCESSORS  p (4, 4)
!HPF$ DISTRIBUTE  a (CYCLIC, CYCLIC)  ONTO  p
```

onto a 2-dimensional processor array. Assuming that the size of array `a` is larger than the size of the processor array, this helps to spread-out the workload.

Incidentally, the question of which of these distributions is best, i.e. results in the shortest execution time, or indeed whether a block distribution is better, is best answered by timing the code for a variety of reasonable distributions. The optimal distribution often depends on a number of factors, such as the problem size, the number of processors, and the characteristics of the target computer and the HPF compiler, as it does in this case. Such experimentation would also be needed to obtain the optimal message-passing program. However, it is much easier to experiment with different data distributions in HPF than in explicit message-passing programs!

3.6 Unspecified mapping

The mapping of any data object may be left unspecified in an HPF program. In particular, it is often not specified for scalars such as `m`, `n` and `r` in the above example.

Although the default mapping is implementation-dependent, on distributed-memory architectures it is likely that data objects without a specified mapping will be '*replicated*', that is, every processor will store a copy of them. That 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 (e.g. as `DO`-loop indices, or in `DO`-loop control expressions, `IF` and `DO WHILE` conditions, etc), or are referenced in specification expressions, array subscripts, 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 'privatised' 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 loop so as to preserve the program's semantics.

4 Alignment

The ALIGN directive relates the elements of a data array to the elements of another array, such that elements that are 'aligned' with each other are guaranteed to be mapped to the same processor(s), regardless of the distribution directives. Thus if an array A is aligned with another array B, the distribution of A is determined by that of B, and only the latter is specified.

Alignment can also be specified for scalar objects. However, for most of this section we will concentrate on array objects, returning briefly to the alignment of scalars at the end.

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. This means 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, for example 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.

The array immediately after the ALIGN keyword is called the *alignee*, and the array after the WITH keyword, with which it is aligned, is the *align target*. In this section we shall use the symbols A and T as shorthand for 'alignee' and 'align target' respectively.

All elements of A are involved in the alignment relation. In the ALIGN directive, A's name is followed by a parenthesised list with an entry for each dimension which must be either:

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

We will consider these three cases in turn.

4.1 Using ':' in an alignee dimension

The simplest case is when all of A's dimension entries are ':'s, which is a standard Fortran 90 way of specifying a whole array.[3] T is then in general a *regular section* of an array, specified

[3] 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.

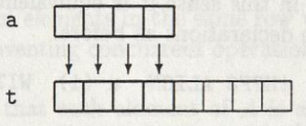
using the normal Fortran 90 subscript triplet notation.[4] Dimensions of A containing a ':' are matched in order with dimensions of T containing subscript triplets, and matching dimensions must have the same number of elements selected.[5] Each element of A is aligned with the corresponding element of T i.e. that in the same position within the regular section.

For example, with the declarations:

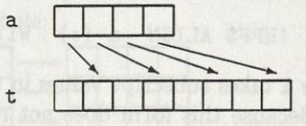
```
REAL  a(4), t(8)
```

some possible alignments are:

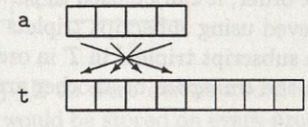
```
!HPF$ ALIGN  a (:)  WITH  t (1:4)
```




```
!HPF$ ALIGN  a (:)  WITH  t (2:8:2)
```



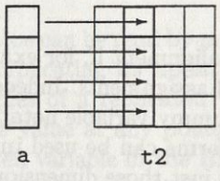

```
!HPF$ ALIGN  a (:)  WITH  t (4:1:-1)
```



As normal for Fortran 90 regular sections, any dimension of T can contain a scalar subscript rather than a subscript triplet. The only requirement is that the regular section selected from T must conform with A.[6] In this way an array can be 'embedded' within a larger dimensional array, as in the following example:

```
!HPF$ ALIGN  a (:)  WITH  t2 (:,2)
```

('a' embedded in 't2')



If `t2` were distributed over a 2-dimensional processor array, then `a` would only be stored on one column of processors, namely that which stores column `t2(:,2)`.

4.2 ALIGN dummy variables

An alternative to using ':' as a dimension entry in A is to use a *dummy variable*, e.g. `i` or `j`. In that case, an expression $f(i)$ that is linear in the dummy variable i can appear in

[4] A subscript triplet has the general form $[l]:[u][:s]$, where l, u and s are scalar integer expressions and $[\ldots]$ 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.

[5] In Fortran 90 terminology, A and T must *conform*, or have the same shape.

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

one dimension of T.[7] Different dummy variables must be used in different dimensions of A, and each dimension of T must not reference more than one dummy variable. Dimensions of A and T involving the same dummy variable are then matched. Matching dimensions of A and T need not be in the same order, unlike the case for subscript triplet notation. An element of A with subscript value i in the dimension concerned is then aligned with an element of T whose subscript value in the matching dimension is $f(i)$. Obviously $f(i)$ must be a valid subscript for T for all i in the subscript range of A. Notice that, because $f(i)$ is linear in i, it generates a regular section when applied to the complete subscript range of i, so in this sense it is equivalent to the subscript triplet notation. For example, with the same declarations as before:

```
!HPF$ ALIGN  a (i)  WITH  t (2*i-1)
```

has the same meaning as:

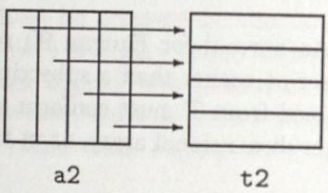
```
!HPF$ ALIGN  a (:)  WITH  t (1:7:2)
```

since `i` takes subscript values in the range `(1:4)`.

Because this form does not require matching dimensions of A and T 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 dimensions of A specified by colons are matched with subscript triplets in T in order of appearance. For example, an array `a2` can be aligned with the *transpose* of another array `t2` as follows:

```
!HPF$ ALIGN  a2 (i,j)  WITH  t2 (j,i)
```

(dimensional permutation)



This is a good alignment if, for example, `a2(i,j)` often appears together with `t2(j,i)` in expressions and assignments. Indeed, dimensional permutation is the only real justification for using the dummy variable notation—otherwise, the triplet notation is clearer and more concise. Both forms can be used in the same directive, so the use of dummy variables can be restricted to just those dimensions necessary for dimensional permutation. For example, the above could equally well be written:

```
!HPF$ ALIGN  a2 (i,:)  WITH  t2 (:,i)
```

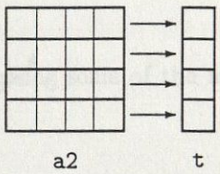
4.3 Collapsing and replication

A '*' may appear as a dimension entry in any dimension of A or T, and it means that the subscript in that dimension plays no part in the alignment relation. Thus, if a '*' appears in one of A's dimensions, it means that each set of elements whose subscripts differ only in that dimension is aligned with the same element(s) of T, which is called *collapsing* a dimension. For example:

[7] In fact the dummy variable i need not be referenced in T's dimension list. If it is not, it is equivalent to a * in A's dimension list (which we explain shortly).

```
!HPF$ ALIGN  a2 (:,*)  WITH  t (:)
```

(2nd dimension of 'a2' collapsed)

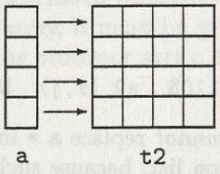


means that `a2(i,j)` is aligned with `t(i)` for all `i` and `j`. In other words, every element in a given row of `a2` is stored on the same processor(s), since each row is aligned with a single element of `t`, which cannot be split across multiple processors however `t` is distributed. This means that 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 one of T's dimensions, it means that each element of A is aligned with the whole set of elements of T whose subscripts differ only in that dimension, i.e. A is copied, or *replicated*, over that dimension of T. For example:

```
!HPF$ ALIGN  a (:)  WITH  t2 (:,*)
```

('a' replicated over 2nd dimension of 't2')



means that `a(i)` is aligned with `t2(i,j)` for all `i` and `j`. If `t2` were distributed over a 2-dimensional processor array, then a copy of element `a(i)` would be stored on every processor in a particular row of the processor array, namely the row of processors over which `t2(i,:)` is distributed. In other words, `a` would be *replicated* over the second dimension of the processor array. More generally, `a` is replicated over the second dimension of `t2`, and consequently it is also replicated over whatever processor array dimension the second dimension of `t2` is distributed over.[8]

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*, that is, 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 large data objects, the storage cost of keeping multiple copies may also be a disadvantage. Nonetheless, this is often a sensible storage strategy for scalar variables, particularly control variables such as `DO`-loop indices. It is also a natural strategy for mapping arrays onto a higher dimensional processor array, and may well be a good distribution for small arrays that are read more frequently than they are written.

Collapsing and replication may be combined. Thus:

```
!HPF$ ALIGN  a (*)  WITH  t (*)
```

means that all elements of `a` are aligned with every element of `t`, that is, `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`.

Incidentally, an alternative to using * in one of A's dimensions is to use a dummy variable that does *not* appear in T's dimension list. Thus

[8] Incidentally, the alignment notation should not be taken too literally in the case of replication. In this example, if the second dimension of `t2` has size 4 and is distributed over 2 processors, the alignment directive suggests that each would store 2 identical copies of `a`. There is an obvious optimisation!

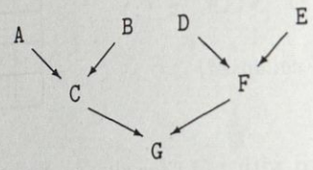


Figure 2: An 'alignment tree'

```
!HPF$ ALIGN  a2 (:,*)  WITH  t (:)
```

could also be written as:

```
!HPF$ ALIGN  a2 (:,j)  WITH  t (:)
```

or

```
!HPF$ ALIGN  a2 (i,j)  WITH  t (i)
```

However, we *cannot* replace a $*$ in T's dimension list by a dummy variable not appearing in A's dimension list, because such an identifier would be interpreted as a subscript signifying embedding. In any case, we recommend the '$*$' form, as its meaning is more easily identifiable.

4.4 Aligning scalars

So far we have concentrated on aligning arrays. Scalars 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). Conversely, an array can be totally collapsed onto a scalar object.[9]

4.5 Alignment trees

Notice that, by 'chaining together' alignments it is possible to create an 'alignment tree' as in Figure 2. 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. For the sake of simplicity we recommend restricting alignments to a depth of one by not further aligning align targets.

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.

[9] However, if the *alignee* is scalar a different form of `ALIGN` directive to that described here must be used. An example is shown in Section 4.6.1. 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.

4.6 Purposes of alignment

We conclude this description of alignment by summarising some of the main reasons for using it.

4.6.1 Expressing more general mappings

Alignment allows more general data mappings to be expressed than can achieved using the DISTRIBUTE directive alone.

For example, it is required in order to describe the mapping of a *regular section* of an array. This is necessary if we want to pass a regular section, e.g. `a (10:20:2)`, as an argument to a procedure, and wish the corresponding dummy argument to have the same mapping as the array section actual argument so that no remapping is performed on entry to and exit from the procedure. We will show an example of this later.

It turns out that alignment must also be used to specify *replication*. There is no way to specify that a variable is replicated using only DISTRIBUTE directives. For example, to specify that a scalar is replicated over a processor array, it must be given a replicated alignment with an array that is then distributed over the processor array, e.g.:[10]

```
      REAL  s,  a (n)
!HPF$ PROCESSORS  p (4)
!HPF$ ALIGN  WITH  a (*) :: s
!HPF$ DISTRIBUTE  a (BLOCK)  ONTO  p
```

If there is no suitable data array to serve as the align target, then we shall see later how to use an HPF *template* for this purpose.

Having said this, if no mapping at all is specified for a variable, many HPF compilers will totally replicate it, i.e. store a copy of the whole variable on every processor, as we said in Section 3.6.

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

```
!HPF$ ALIGN  a2 (:,*)  WITH  t (:)
!HPF$ DISTRIBUTE  t (BLOCK)  ONTO  p
```

is equivalent to:

```
!HPF$ DISTRIBUTE  a2 (BLOCK, *)  ONTO  p
```

4.6.2 Expressing intended relationships between distributions

It is clearer and safer to explicitly align objects that are intended to be distributed in a related manner, rather than relying on alignment being achieved as a side-effect of distribution.

[10] Here we have used a different syntactic form for the ALIGN directive. This form must be used when the alignee is a scalar. This quirk arises from the need to ensure that HPF directives are unambiguous if blanks are insignificant, as they are in Fortran's fixed source form. This alternative syntax can actually be used in all cases, and is perhaps better as it corresponds to Fortran 90's new syntax for declarations.

4.6.3 Simplifying performance tuning

The optimal alignment is largely determined by the program, i.e. by which array elements tend to be accessed together in operations and assignments. Therefore it tends to be an invariant property of the program. In contrast, the optimal distribution may depend on a number of external factors, such as the problem size, the number of processors and the characteristics of the target computer. Therefore, if the two-level mapping of alignment and distribution is used, tuning a program for different architectures should involve changing only its `DISTRIBUTE` directives and not its `ALIGN` directives.

5 Mapping of procedure arguments

We shall now consider how HPF handles data mapping across procedure boundaries. We start by briefly reminding readers of some Fortran terminology that we shall use.

A *procedure* is a function or a subroutine. In the *definition* (i.e. the body) of a procedure, its arguments are called *dummy arguments*; they are normally named variables or dummy procedure names. When a procedure is referenced, the arguments that are passed to it by the caller are called its *actual arguments*; they may be variables, expressions or actual procedures. Finally, Fortran 90 introduced the concept of an *explicit interface* into Fortran. This means that the caller of a procedure is provided with complete information about the procedure's dummy arguments and, for a function, its result, for example, their types, shapes, whether they are used as input and/or output arguments, etc. The interface is automatically explicit for intrinsic, internal and module procedures, and can be made explicit for external procedures by declaring an '`INTERFACE` block' that contains the required information. For example, the following is an `INTERFACE` block for the subroutine `Gauss_itn` in Figure 3:

```
INTERFACE
  SUBROUTINE  Gauss_itn (matrix, col, row, elem, n1, n2)
    INTEGER, INTENT (IN)    :: n1, n2
    REAL,    INTENT (INOUT) :: matrix (n1, n2),  row (n2)
    REAL,    INTENT (IN)    :: col (n1),  elem
  END SUBROUTINE
END INTERFACE
```

The basic principles governing the mapping of procedure arguments in HPF 2.0 are quite simple. They can be expressed as follows:

- The mapping of a procedure's *dummy* arguments can be specified in the same way as for local and global variables, using the directives we have already described.
- When a procedure is referenced, its *actual* arguments can have a different mapping from the dummy arguments with which they are associated. However, if any actual argument has a different mapping from the corresponding dummy argument, then the interface of the procedure must be explicit in the caller, and the interface must specify the mapping directives for the dummy arguments.

In HPF, if the interface of a called procedure is explicit in the caller, and any actual argument is not mapped as specified in the interface, then it is automatically copied to a

temporary variable with the required mapping just before entry to the procedure to satisfy the dummy argument's mapping directives, and copied or remapped back on return.[11] Within the called procedure, therefore, the dummy arguments are assumed to already have the mapping specified for them, and no remapping is necessary.

However, if a called procedure *does not have an explicit interface*, then none of its actual arguments are remapped since their required mapping is not known. Therefore the actual arguments must already have the same mapping as the dummy arguments or the program will be erroneous. Hence it is safest to ensure that the interface of a called procedure is explicit, unless one can be certain that no argument remapping is required.

We emphasise 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.

There are several reasons why HPF allows a data object to be remapped when it is passed as an argument to a procedure. The most obvious is that a dummy argument may be associated 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.

5.1 An example of argument mapping

We shall demonstrate dummy argument mapping using a modified version of the Gaussian elimination code, shown in Figure 3, in which the update in each iteration is performed by calling a subroutine `Gauss_itn`. Admittedly this is a somewhat artificial example as it probably is not worth calling a subroutine to perform just two assignments, but it will serve for illustration.

The dummy arguments of `Gauss_itn` could be mapped as follows:

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

Dummy argument `matrix` is associated with the actual argument `a(r+1:m, r+1:n)`, which is a *regular section* of an array. Suppose that a is distributed `(BLOCK,BLOCK)` in the caller. Then in general `a(r+1:m, r+1:n)` occupies only a subset of the processors, namely 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* processor array, as described in Section 3.2. To acquire this mapping, `a(r+1:m, r+1:n)` must generally be copied to a temporary array with the required mapping before entry to `Gauss_itn`, and copied back on return. For this to happen, the interface of `Gauss_itn` must be explicit in the caller.

[11] ... unless the compiler can determine that one or other of these operations is unnecessary. For example, a dummy argument declared as `INTENT (OUT)` need not be copied in on entry, and one declared as `INTENT (IN)` need not be copied back on return.

```
INTEGER  m, n, r
REAL     a (m,n)
...
DO r = 1,m                                                          &
  CALL Gauss_itn (a(r+1:m, r+1:n),
                  a(r+1:m, r), a(r, r+1:n), a(r,r), m-r, n-r)

END DO
...
SUBROUTINE  Gauss_itn (matrix, col, row, elem, n1, n2)
  INTEGER, INTENT (IN)    :: n1, n2
  REAL,    INTENT (INOUT) :: matrix (n1, n2),  row (n2)
  REAL,    INTENT (IN)    :: col (n1),  elem

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

Figure 3: Gaussian elimination with each iteration done by a subroutine call

In many HPF implementations, the value assigned to an array element is computed by the processor(s) that 'own(s)' it, that is, 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(r+1:m, r+1: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 before and after the call to `Gauss_itn`, so it is uncertain whether the remapping would reduce 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 3.5.1, since then the section `a(r+1:m, r+1: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. We will see later how to describe the dummy argument mappings so that no remapping occurs.

`col`, `row` and `elem` are aligned with `matrix`. The alignment of `col` means that it is block distributed over the first dimension of processor array `procs` and replicated over the second dimension of `procs`, so each column of the processor array stores a complete copy of `col`. Similarly, the other `ALIGN` directives mean that `row` is replicated over the first dimension of `procs`, so each row of `procs` has its own copy of `row`, and `elem` is replicated over both dimensions of `procs`, so every processor has its own copy of `elem`.

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.

We mentioned in Section 4.3 that all copies of a replicated variable must be updated, since they must be kept 'consistent'. 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 specified mapping and so must be copied into and out of `Gauss_itn` (though an good implementation would not copy back `col` and `elem` as they are declared to be `INTENT (IN)`, meaning that they are not updated). Again, this means that the interface must be explicit in the caller.

5.2 The INHERIT directive

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

which means that the associated actual arguments can have *any* mapping and will not be remapped—even if they are array elements or sections. In general, the compiler will generate code to handle any mapping for the arguments (unless it can somehow determine the possible actual argument mappings).

Some dummy arguments may have inherited mapping while others have explicit 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.

It should be noted that while `INHERIT`ed mapping avoids any remapping of the actual arguments on entry and exit, it can result in less efficient code inside the procedure as there may be little or no information about the dummy arguments' mapping at compilation time.

5.3 Transcriptive distribution

It is possible for a dummy argument to inherit some, but not necessarily all, of the actual argument's distribution characteristics 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 processors arrangement over which it is distributed, are inherited from the actual argument. Clauses in a DISTRIBUTE directive consisting of just asterisks are called *transcriptive*.

However, there is a difference between the above directive and the INHERIT directive, because the above form is only legal if the actual argument corresponding to matrix has a mapping that can be described using a DISTRIBUTE directive. It would not be legal in the example of Figure 3, since there the actual argument is a *regular section* of an array, whose mapping can only be described using alignment, as explained in Section 5.1. In this sense the INHERIT directive is more general than the above form.

In a DISTRIBUTE directive, one clause can be transcriptive while the other is specified. For example:

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

means that matrix inherits its distribution format but is distributed over a specified processors arrangement p, that is, 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  matrix (BLOCK, BLOCK)  ONTO  *
```

means that matrix is to be block distributed onto whatever processors arrangement the actual was distributed onto.

In our experience transcriptive distribution is not often used.

5.4 HPF-1 rules for argument mapping

We briefly mention that HPF-1 had different rules concerning the mapping of procedure arguments.

In HPF-1, explicit interfaces were *not* required if the actual argument had a different mapping to the dummy argument. Therefore, when compiling a procedure, an HPF compiler could not assume, as it can in HPF 2.0, that its dummy arguments were already mapped as specified, and would typically generate code to test the mapping and perform any necessary remapping *inside* the procedure. Therefore, as an optimisation, HPF-1 provided a special *descriptive* form of the mapping directives for dummy arguments, which asserted that the corresponding actual arguments already had the specified mapping so no tests or remapping would be performed. The descriptive form was distinguished by having extra asterisks in various places; see [12] or [15] for details.

The descriptive form is retained in HPF 2.0 for compatibility, but its meaning has changed. It is now a request that a warning be generated at compile time or runtime if an actual argument needs to be remapped to satisfy the dummy argument's mapping directives. In HPF 2.0, an explicit interface is still mandatory if argument remapping is required, even if descriptive directives are used. We recommend that new users of HPF do not concern themselves with or use the descriptive form.

Incidentally, it should be noted that some correct HPF-1 codes may no longer work under HPF 2.0. This happens if the code does not have explicit interfaces for calls that require argument remapping.

Another change is that HPF-1 allowed a dummy argument to appear in both an INHERIT *and* a DISTRIBUTE directive. The meaning of this combination of directives was fairly hard to comprehend! It is now disallowed; INHERIT and DISTRIBUTE are now mutually exclusive.

6 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, for example:

```
!HPF$ TEMPLATE  s,  t (16),  u (2*n+1, 2*n+1)
```

Like processor arrangements, templates 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 TEMPLATE directive to be visible in several program units is to declare it in a module which is USEd by the program units. Otherwise, templates must be declared locally in every program unit in which they are used.

The sole function of a template is to provide an abstract object with which data objects can be aligned and which can then be distributed, that is, to provide an intermediary in the mapping of data objects to abstract processors.

For example, suppose that in the Gaussian elimination example of Figure 3, array a is distributed as follows:

```
!HPF$ PROCESSORS  procs (4,4)
!HPF$ DISTRIBUTE  a (CYCLIC, CYCLIC)  ONTO  procs
```

Sections of a are passed as actual arguments to subroutine Gauss_itn, where they are associated with dummy arguments matrix, col, row and elem. Further suppose that we wish the dummy arguments to be declared with the same mappings as the actual arguments so that no remapping occurs. This can be done using ALIGN directives as shown in Figure 4. To help describe the mapping we have modified Gauss_itn's argument list slightly, passing in 3 arguments m, n and r rather than the 2 array size arguments n1 and n2 used before.

Notice that the use of a template is almost indispensable in this example. The dummy arguments are associated with *regular sections* of array a, so their mappings can only be described by aligning them with equivalent regular sections of an array with the same dimensions as a. (That is, they cannot be described by DISTRIBUTE directives alone.) 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. Notice also that both of these 'solutions' involve modifying the actual Fortran source code, as opposed to just adding directives to it, solely for the purpose of expressing data mapping—something that is better avoided. 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.

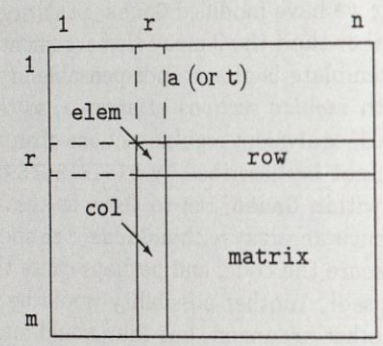
An alternative mapping for the dummy arguments is the following:

```
!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 (*, *)
```

In this case matrix has the same mapping as the actual argument that is passed to it, whereas dummy arguments col, row and elem are replicated over matrix as they were in

```
      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$   PROCESSORS  procs (4,4)
!HPF$   DISTRIBUTE  t (CYCLIC, CYCLIC)  ONTO  procs
!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  WITH  t (r, r) :: elem
        ...
```

(a): Code



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

Figure 4: Mapping Gauss_itn's dummy arguments to have the same mapping as its actual arguments

Section 5.1, which implies that their corresponding actual arguments have to be remapped on entry and exit, but avoids communications within the body of `Gauss_itn`. Therefore the interface must be explicit in this case.

6.1 Purposes of templates

We conclude our description of templates by summarising some of the reasons for using them.

1. A template is indispensable for providing a virtual array with which to align data arrays when there is no suitable data array to serve that purpose, as in the last example.

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

   For example, 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. By contrast, when arrays are aligned with other arrays an arbitrarily complicated alignment tree can be constructed (see Section 4.5), which can make it difficult to identify the 'root' object with which a given array is ultimately aligned.

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

3. 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.

4. In practice we have found that a convenient technique for specifying data mapping is to align everything (that is explicitly mapped) with a single template that is declared and `DISTRIBUTE`d in a separate file. This file is then `INCLUDE`d in all program units that have data mapping directives.[12] Then there is only one `DISTRIBUTE` directive in the whole program, and to experiment with different distributions merely requires changing this one directive, which is easier and safer than changing `DISTRIBUTE` directives scattered throughout the code!

7 Restrictions on sequence and storage association

HPF 2.0 imposes an important restriction on variables that are explicitly mapped (i.e. aligned or distributed), which is that they may not be used in ways that depend on Fortran 77's model of sequence and storage association. More precisely, the following Fortran 77 practices are disallowed for mapped variables:

[12] A more modern way of achieving the same effect is to use a module rather than an include file.

1. Passing an array element as an actual argument corresponding to a dummy array.

2. Having a shape or size mismatch between an array actual argument and the corresponding dummy array, e.g. passing an array with shape (100) and receiving it as an array of shape (10,10) or shape (40).

3. Declaring an array as assumed size, i.e. with a '*' in the final dimension of its shape specification, or associating an actual argument with an assumed size dummy array.

4. Belonging to a storage area of a common block that is partitioned differently, or declared with a different shape, in different procedures. For example in:

```
SUBROUTINE s1
  COMMON /com/ a (100), b (100), d (2,5), e (5,5)
  ...
END
SUBROUTINE s2
  COMMON /com/ c (200), d (10), e (5,5)
  ...
END
```

   the variables `a`, `b` and `c` cannot be explicitly mapped because they belong to a storage area that is partitioned into variables differently in the two subroutines. `d` cannot be explicitly mapped as it has a different shape in different subroutines. `e` *can* be explicitly mapped however.

5. Using `EQUIVALENCE` to partition a particular storage area into variables in different ways, or with different shapes. For example in:

```
REAL  a (100), b (100), c (200)
EQUIVALENCE (a, c), (b, c(101))
```

   the variables `a`, `b` and `c` cannot be explicitly mapped for the same reasons as above.

Variables subject to any of these practices are called 'sequential' in HPF, and they cannot be explicitly mapped. Cases 1–3 imply that both the dummy and the actual arguments are sequential. Strictly speaking, sequential variables should also be declared as such using the HPF `SEQUENCE` directive, e.g.:

```
!HPF$ SEQUENCE :: a,b,c
```

in cases where the HPF compiler cannot determine by local analysis (i.e. by analysis local to the program unit) that they are sequential; that is for dummy arguments in cases 1–2, for actual arguments in cases 1–3 unless there is an explicit interface, and in case 4. This is for the benefit of HPF implementations in which the default mapping in the absence of explicit mapping directives is *not* replicated.

Note that these practices are the basis for a technique that is widely used in 'dusty deck' Fortran codes to simulate dynamic memory allocation, which was not directly supported in Fortran 77. The technique is to declare a single large array, often called a '*work array*', and then 'carve it up' dynamically into separate variables. It is 'carved up' by, for example, passing an element of it as an argument to a procedure and receiving it as an array of

the required shape, whose storage therefore occupies a segment of the work-array, or by declaring it in a common block that is partitioned and used differently in different program units. Unfortunately, if these practices are used, the 'work array' and the variables it is partitioned into cannot be distributed.

Fortran 90 provides the required dynamic memory management features by means of automatic arrays, `ALLOCATABLE` arrays and `POINTER`s, so these practices are no longer required. However, modifying code to replace work arrays by dynamic arrays can be quite a large task. In our experience this is often the most time-consuming part of porting a large dusty deck Fortran code to HPF. However, the effort is probably worthwhile just for the improved legibility and maintainability of the code, quite apart from anything else.

7.1 Other issues

This concludes our description of HPF data mapping. There are a few issues that we have not covered, for example, the mapping of allocatable arrays, pointers or derived type components. For full details the reader is referred to [19].

8 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, HPF 1.0 introduced 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.

In fact, HPF's `FORALL` and `PURE` features have been incorporated in the new revision of the official Fortran standard, Fortran 95 [18]; indeed, they are the main additions to the language at this revision. Nevertheless, we describe them here since readers may not be familiar with them yet, and also because HPF imposes some restrictions on data mapping in `PURE` procedures which are irrelevant in the context of normal Fortran and are therefore omitted from the Fortran standard.

8.1 `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. For example:

```
FORALL (i=1:10)  a(i) = b(i) + c(i+2)
```

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 above 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 neighbours. 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 paper 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 $[\ldots]$ denotes an optional item. l, u and s must not depend on FORALL indices, so the 'index space' is rectangular (although this can select non-rectangular array sections as we shall see shortly).

A FORALL assignment need not be scalar—it can be an array assignment. Furthermore, subscripts 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(indx(i)) = b(i)
```

is legal only if indx contains no repeated values.

```
FORALL (i=0:9, j=1:5)  a(10*i+j) = c(j)
```

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 sometimes 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 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:

- 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/))
  ```

- To avoid the conformance rules for array assignments. For example, the following array assignment from the Gaussian elimination code of Figure 1 requires the use of the SPREAD intrinsic function so that all array sections conform:

  ```
  a(r+1:m, r+1:n) = a(r+1:m, r+1:n) - (SPREAD (a(r+1:m, r), 2, n-r)  &
                                      * SPREAD (a(r, r+1:n), 1, m-r))
  ```

  It can be expressed more simply using a FORALL:

  ```
  FORALL (i=r+1:m, j=r+1:n)  a(i,j) = a(i,j) - a(i,r) * a(r,j)
  ```

- 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) == 0)
  ```

- To express non-rectangular 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
  ```

- 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 3-dimensional 'irregular section' from the 2-dimensional array a. Fortran 90 vector subscript notation could only form a 1 or 2-dimensional section from a, in which ivec and jvec each depend 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 1 each time, and the results are accumulated into C, the matrix product $C = AB$ is produced.

- Finally, a FORALL statement must be used when the constituent elemental assignment involves a reference to a non-elemental 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 non-elemental 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. 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.

For completeness we mention that an assignment in a FORALL statement or construct may be a pointer assignment rather than a normal assignment.

Incidentally, because of the generality of access patterns, and therefore communication patterns, that can be expressed using FORALL, there is no guarantee that any particular FORALL will be executed in parallel. Which ones are executed in parallel depends on the capabilities of the HPF implementation used. Those that are not executed in parallel will be 'sequentialised', i.e. their instances are executed in sequence, so they will still be evaluated correctly. However, we recommend using standard Fortran 90 array syntax rather than FORALL where possible, since this is more likely to be implemented efficiently and in parallel.

8.2 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 *non-deterministic*. 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 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.

For this and other reasons (which we explain later) it is forbidden to reference ordinary user-defined functions in a FORALL assignment or mask expression.[13]

However, HPF-1 introduced a new class of functions called '*pure*' functions, which are guaranteed to be side-effect free and which can be used in these contexts. 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. A pure function must not contain any operation that might conceivably change the value or pointer association of a dummy argument or global variable ([12] gives a full list of disallowed operations), or SAVE local variables, or reference non-pure 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:

- 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 any of these contexts, its interface must be explicit, and both its interface and definition must specify the PURE keyword and the INTENT[14] of its non-pointer and non-procedure dummy arguments (though admittedly this is redundant for a pure function as its arguments must be INTENT(IN) by definition). Intrinsic functions, including HPF intrinsic functions, are always pure and require no explicit declaration of this

[13] 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.

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

```
      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 5: Using a PURE function to plot the Mandelbröt set.

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.

8.2.1 Functional parallelism

As an example of the use of pure functions, Figure 5 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.[15] Thus in Figure 5, 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

[15] Of course, SIMD architectures cannot fully exploit this potential.

parallelism can also arise via 'independent' DO-loops and 'extrinsic' procedure references, both of which are described later.

8.2.2 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; 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 behaviour requires some degree of shared-memory support, either in hardware or software.

For efficiency the calling program unit should have the freedom to choose the processor subset on which to execute any particular pure procedure reference, for example to maximise 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 non-shared-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 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.[16]

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 6. 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 2-dimensional processor array. Incidentally, in this example update_row is a Fortran 90 '*internal*' function, whose interface is automatically explicit in the caller.

[16] However, the implementation on non-shared-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.

```
      INTEGER  m, n, r
      REAL     a (m,n)
!HPF$ DISTRIBUTE  a (CYCLIC, CYCLIC)
      ...
      DO r = 1,m
        a(r, r+1:n) = a(r, r+1:n) / a(r, r)
        FORALL (i=r+1:m)                                    &
          a(i,r+1:n) = update_row (a(i,r+1:n), a(i,r), a(r,r+1: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 6: Gaussian elimination with each row updated by a pure function call

An efficient implementation of the FORALL might broadcast row `a(r, r+1:n)` so that it is aligned with every row `a(i, r+1:n)`, $i > r$, according to the alignment specified in the pure function[17], and then execute each instance i of the FORALL on the processors that own the relevant assignment variable (and argument) `a(i, r+1: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, r+1: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, r+1:n)`—thus sequentialising 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, r+1: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 a manner that is efficient for the operations performed within the function.

8.3 INDEPENDENT directive

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

[17]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.

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 [19], section 5.1. Unlike the case for PURE procedures, these are assertions about *behaviour*, 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, since 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 can also have optional 'NEW' and/or 'REDUCTION' clauses (the latter of which was introduced in HPF 2.0.) The NEW clause specifies 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. The REDUCTION clause specifies that certain variables are used to accumulate a value under a 'reduction' operation, e.g. +, with each iteration of the DO-loop contributing one term to the accumulated value.

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 non shared-memory architectures, we recommend the use of array or FORALL syntax rather than independent DO-loops where 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 synchronisation 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 optimisation.

As with all directives that provide 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.

9 Extrinsic procedures

An HPF program can call procedures written in another programming language and/or using a different programming model. Such procedures are called *extrinsic* procedures. 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 parallelism that cannot be achieved in HPF, or to hand-tune critical kernels, etc.

When an extrinsic procedure is called from HPF its interface must be explicit, and it must specify:

EXTRINSIC (*extrinsic-spec*)

before the FUNCTION or SUBROUTINE keyword in its header statement. *extrinsic-spec* specifies the procedure's programming language and/or programming model. It is a list containing any or all of the following specifiers:

LANGUAGE = *char-string*, where *char-string* can be 'HPF', 'FORTRAN', 'F77', 'C' or an implementation-dependent value;

MODEL = *char-string*, where *char-string* can be 'GLOBAL', 'LOCAL', 'SERIAL' or an implementation-dependent value; and

EXTERNAL_NAME = *char-string*, where *char-string* is the procedure's name in the *extrinsic* programming language, in case it is different from the name by which it is referenced in HPF.

Either the language or the model *must* be specified. The keywords LANGUAGE =, MODEL = and EXTERNAL_NAME = can be omitted if they are specified in the above order. The combination of an extrinsic programming language and programming model is called an '*extrinsic kind*'.

As an alternative to the above syntax, the *extrinsic-spec* can consist of just a name (not in quotes) which serves as a shorthand for a particular extrinsic kind. The HPF 2.0 language specification defines the following extrinsic kind names:

HPF: short for LANGUAGE='HPF',MODEL='GLOBAL'. This means HPF itself, and is the default for a procedure compiled by an HPF compiler.

HPF_SERIAL: short for LANGUAGE='HPF',MODEL='SERIAL'.

HPF_LOCAL: short for LANGUAGE='HPF',MODEL='LOCAL'.

F77_LOCAL: short for LANGUAGE='F77',MODEL='LOCAL'.

The Approved Extensions part of the HPF 2.0 language specification defines the rules and semantics of the extrinsic models 'SERIAL' and 'LOCAL', and the extrinsic kinds HPF_SERIAL, HPF_LOCAL, F77_LOCAL and 'C'. However, it does *not* stipulate which, if any,

extrinsic kinds must be supported by an HPF implementation. Indeed, an HPF implementation is free to define its own extrinsic kinds.

The HPF_SERIAL and HPF_LOCAL extrinsic kinds in particular have proved useful and are already widely supported by HPF compilers. Therefore we shall concentrate on them in the rest of this section.

9.1 HPF_SERIAL

EXTRINSIC (HPF_SERIAL) means that the procedure is written in standard Fortran without any HPF mapping directives, and it is called on just *one* processor, which we shall call the 'active' processor. Currently HPF does not provide a way to specify which processor that is. The procedure's variables and dummy arguments are all stored locally on the active processor.

When an HPF_SERIAL procedure is called from HPF, if any of its actual arguments are distributed they are remapped so that they are stored entirely on the active processor before entry, and their original distributions are restored after return.

Therefore, the code in an HPF_SERIAL procedure is executed just like sequential Fortran running on a single processor. Indeed, single processor sequential execution is the essential meaning of the 'SERIAL' extrinsic model, regardless of the extrinsic programming language. Thus the SERIAL model can be regarded as a means of interfacing sequential code, which may be in a different programming language, to HPF.

9.1.1 Some uses of HPF_SERIAL

This suggests that one use for HPF_SERIAL is to 'package' Fortran code that forms part of an HPF application but which one does not want to convert to HPF (or at least, not yet). We shall call such code, i.e. code that does not have any distributed variables, '*serial*' code.

Note that it is not really necessary to package serial code into HPF_SERIAL procedures. It can be treated simply as HPF code that does not have any distributed variables. Then all of its variables will be replicated, i.e. stored in their entirety on every processor, and it will be executed identically on every processor, rather than on just one processor as in the case of HPF_SERIAL. In most cases the HPF version will not perform any communications since all of its variables are stored locally, so its execution time should be similar to that of an HPF_SERIAL version, which should in turn be similar to that obtained if the code were compiled as standard Fortran. However, in practice there may be some overheads in the code generated by an HPF compiler even without communications, since, for example, it may have to maintain extra information about data distribution (even for replicated variables). Hence an HPF_SERIAL version may be slightly faster than an HPF version of a serial code.

Furthermore, there is at least one situation in which serial code *can* give rise to communications when compiled as HPF. That is if it performs input, e.g. in READ statements.[18] In many HPF implementations I/O is performed by a single processor, so a READ statement that defines a replicated variable would cause that processor to input its value and then broadcast it to all the other processors. This broadcast is avoided if the READ statement is in an HPF_SERIAL procedure, since then it is executed on just one processor so there are no other processors to communicate with.

Although there are no communications *inside* an HPF_SERIAL procedure, one should note that if it is called from HPF, the HPF caller may have to perform communications before

[18] I/O related communications can also occur in less obvious ways, e.g. when a STATUS = variable is defined in a WRITE statement.

entry to remap distributed input arguments[19] so that they are stored entirely on the active processor, and again when the procedure returns to restore its output arguments to their original mapping, e.g. by broadcasting them if they are replicated. Hence communications inside the procedure are replaced by communications on entry and exit.

These considerations suggest a situation in which HPF_SERIAL may be used as an optimisation. That is if a section of code performs many 'small' READ instructions, e.g. by inputting a value to each element of an array individually. It is better to replace this by a few 'large' READs if possible, e.g. by inputting to the whole array at once, since a few 'large' communications are usually faster than many 'small' ones. However, if that is not possible an alternative may be to put the READ statements in an HPF_SERIAL procedure so that they do not cause any communications. Then the communications associated with the READ are replaced by communications of the procedure's output arguments, which are likely to be whole arrays rather than individual elements. This modification will be beneficial if it results in a net reduction in communication time that is not offset by an increased calculation time due to loss of parallelism.

Before leaving the subject of HPF_SERIAL we should say something about how to compile it and how to call other procedures from it. The HPF 2.0 language specification is quite vague on these points, and, perhaps partly because of this, to a large extent they are implementation-dependent. It *does* say that a called procedure has the same extinsic kind as the program unit that calls it, unless there is an explicit interface that specifies otherwise. Thus procedures called from HPF_SERIAL are assumed to be HPF_SERIAL themselves unless otherwise specified. Another point about which we can be certain is that if a procedure is declared to be EXTRINSIC (HPF_SERIAL) in its *definition* (as opposed to in an interface), then it should be compiled by an HPF compiler, one obvious reason being that a normal Fortran compiler would not undertand the EXTRINSIC keyword! However, one would expect the HPF compiler to compile it as 'standard' Fortran without any HPF-specific overheads.

If an HPF_SERIAL procedure calls a large amount of other Fortran code, then one would probably prefer to leave that code unchanged and compile it as normal Fortran, rather than adding EXTRINSIC (HPF_SERIAL) prefixes to every function and subroutine statement and compiling it with an HPF compiler. The portable way to do this would be for the 'top level' HPF_SERIAL routine, i.e. the one called directly from HPF, to declare the routines it calls to be F77_SERIAL or FORTRAN_SERIAL, but this extrinsic kind may not be supported. Indeed, if it *is* supported then it can be used for the 'top level' routine as well. Anyway, in practice many HPF implementations allow a routine that is declared in an interface to be HPF_SERIAL to be defined and compiled as a normal Fortran routine. But this is not guaranteed to work—it is implementation-dependent.

9.2 HPF_LOCAL

Like HPF_SERIAL, EXTRINSIC (HPF_LOCAL) also means that the procedure is written in standard Fortran without any distributed variables. However, in this case it is called concurrently on *all* processors. The variables in an HPF_LOCAL procedure are local to the processor running that particular instance of the procedure, and they may store different values on different processors. This is unlike the situation in HPF, where conceptually there is just one copy of each variable, and if multiple copies of it are stored on different processors then they all have same value.

[19] In practice, many compilers treat an argument as an 'input' argument if it not declared as INTENT (OUT), and as an 'output' argument if it is not declared as INTENT (IN).

Arguments are passed to an HPF_LOCAL procedure exactly as they are to an HPF procedure. This means that if an argument is a distributed array, only the locally-stored segment of it is passed to the procedure on each processor. However, unlike the situation in HPF, the corresponding dummy argument within the HPF_LOCAL procedure represents just this locally stored segment, *not* the whole array that is accessible on the HPF caller side. The subscripts of such a dummy array are with respect to the local segment, as though the local segment were the entire array. In general, code within an HPF_LOCAL procedure can only reference locally stored variables. There is no way to directly reference variables stored on other processors (though they may be accessible indirectly by some implementation-dependent means, e.g. by message-passing).

The interface and body of an HPF_LOCAL procedure *can* contain HPF mapping directives for its dummy arguments, though not for its local or global variables. In an *interface*, such mapping directives have the same meaning as for a normal HPF procedure, that is, they specify the mapping required of the actual argument on entry to the procedure. If the actual argument has a different mapping, the HPF caller remaps it prior to entering the HPF_LOCAL procedure to satisfy the mapping directives, and restores its original mapping on return. In the procedure *body*, mapping directives are understood to describe the mapping that the *actual argument* has on entry to the procedure. Note that they do not refer to the mapping of the *dummy argument*, since that is an entirely locally stored object, namely the local segment of the actual argument. In fact, mapping directives are redundant in the body of an HPF_LOCAL procedure. They are permitted simply to allow consistency between the procedure's definition and its interface[20].

Notice that if an actual argument to an HPF_LOCAL procedure is a distributed array, then in general it has a different size to the corresponding dummy argument, which is just the local segment of the distributed array. Therefore if the dummy argument is declared with an explicit size, that size must be wrong either in the procedure body (if it is the size of the actual argument) or in the interface (if it is the size of the local segment). To avoid this problem, dummy argument arrays in HPF_LOCAL procedures must be declared as assumed shape, e.g. `D (:,:)`, if they are associated with *distributed* actual arguments.

A module called HPF_LOCAL_LIBRARY is provided for use in HPF_LOCAL. It contains routines for enquiring about the shape and mapping of the actual arguments that are associated with an HPF_LOCAL procedure's dummy arguments, for translating between the local and global subscripts of such arguments, and for finding the coordinates of the local processor.

To summarise, HPF_LOCAL, and in fact, the 'LOCAL' extrinsic model generally, allows the use of explicitly local code, that is, code that will be run on each processor exactly as it is written. In the jargon of parallel processing, this is called SPMD ('Single Processor, Multiple Data') code. This kind of programming obviously gives much more control over exactly what happens where than HPF does.

One important use of HPF_LOCAL is to invoke message-passing code from HPF, but it has other uses too. We shall now briefly describe two applications of it that do not involve message-passing.

9.2.1 Coarse Grained Parallelism

HPF was designed primarily to express *fine-grained* data parallelism, whereby operations can in principle be performed on every element of an array in parallel if there are enough

[20]... and also to allow for situations when the procedure definition itself provides the interface, e.g. for module procedures.

```
      INTEGER :: n
      TYPE (struc) :: structs (n)
!HPF$ DISTRIBUTE (BLOCK) :: structs
      INTERFACE
        EXTRINSIC (HPF_LOCAL) SUBROUTINE  calc_struct (structs)
          TYPE (struc) :: structs (:)
        END SUBROUTINE calc_struct
      END INTERFACE
      ...
      CALL calc_struct (structs)
```

Figure 7: Using an HPF_LOCAL procedure for coarse-grained parallelism

processors.

However, for some applications *coarse-grained* data parallelism is more suitable. Such applications typically contain a set of 'structures' of some kind, e.g. a set of grids, each of which can be processed in parallel with the others, although the computations within each structure may be performed sequentially. Examples include computational fluid dynamics and structural mechanics codes that perform simulations on complicated physical domains which are divided into a set of more regular subdomains to simplify the calculations. The subdomains may have various sizes and require various amounts of computation, and perhaps even totally different computations, so this type of application may not be readily amenable to fine-grained data parallelism across the different subdomains.

Figure 7 shows how HPF_LOCAL can be used to obtain coarse-grained data parallelism. `structs` is an array of a derived type, each element of which contains the data for one structure. The array is distributed, so each processor stores one or more elements, and thus the data for one or more structures. It is passed to an HPF_LOCAL routine `calc_struct`, which performs the computations on the structures. Therefore each processor performs the computations for the structures that it stores, independently of the rest. The computations within each structure are performed sequentially, but in parallel with the computations for structures on other processors.

If different structures require different amounts of work then 'load balancing' may be a problem: some processors may have less work to do than others, and thus spend some of their time idle. Hence it may be important to allocate structures to processors in a way that evens out the workload (assuming that there are more structures than processors). With a little ingenuity the above scheme can be modified to allow different numbers of structures to be allocated to each processor, e.g. by distributing an array, each element of which is an allocatable array of structures.

9.2.2 Parallel I/O

As we explained in section 9.1.1, many HPF implementations arrange for external I/O to be conducted via a single processor, which does all of the I/O operations and gathers data from or distributes it to the other processors as necessary. This means that I/O can give rise to a lot of communications and tends to become slower with an increasing number of processors.

Some parallel computers have a parallel filesystem, which allows each processor to per-

```
EXTRINSIC (HPF_LOCAL) SUBROUTINE read (unit,x)
  INTEGER :: unit
  REAL    :: x (:)
  READ (unit) x
END
EXTRINSIC (HPF_LOCAL) SUBROUTINE write (unit,y,z)
  INTEGER :: unit
  REAL    :: y, z (:)
  WRITE (unit) y, z
END
```

Figure 8: Using HPF_LOCAL for parallel I/O

form I/O to its own local disk in parallel with the others. If this capability exists it is clearly advantageous to use it, both to avoid communications within the program and to spread-out traffic on the machine's I/O network.

This can be done simply by enclosing the I/O statements in HPF_LOCAL procedures, as shown in Figure 8. Then each processor READs and WRITEs its locally-stored segment of the data to its own file—a different file for each processor. This involves no communications in the program, and if there is a parallel filesystem it can be done in parallel by every processor. For this to work, all I/O to a particular unit, including OPEN and CLOSE statements, must be enclosed in HPF_LOCAL procedures. Note that this technique involves splitting-up the I/O across multiple files, one per processor, which may not always be convenient. It is probably most useful for I/O to temporary files that are used only during the course of a run.

When OPENing a file in HPF_LOCAL, we recommend that it is given a processor-dependent filename, as shown in Figure 9. This will make it portable to systems where multiple processors share the same filesystem. Without this refinement the program would open multiple files with the same name, which would be erroneous if they share a common filesystem.

10 HPF intrinsic functions and standard library

The final extensions in the HPF 2.0 base language are three new intrinsic functions and an HPF 'standard library'. Due to lack of space we shall simply list these procedures here. The reader is referred to Section 7 of [19] for a detailed description of their arguments and use.

10.1 HPF intrinsic functions

HPF defines a new class of intrinsic functions called 'system enquiry' functions. There are two functions in this class: NUMBER_OF_PROCESSORS, which returns the total number of processors (or in some implementations *processes*) on which the program is running, and PROCESSOR_SHAPE, which returns the system-defined logical shape of the processor array formed by these processors (or processes). On many platforms the processors are not regarded as being configured into an array, in which case PROCESSOR_SHAPE will probably return a value such as (/NUMBER_OF_PROCESSORS()/), meaning they are treated as a 1-dimensional array. PROCESSORS_SHAPE() can also take an optional DIM argument. In HPF

```
      INTEGER :: pid (NUMBER_OF_PROCESSORS)
!HPF$ DISTRIBUTE pid (BLOCK)
      ...
      DO i = 1, NUMBER_OF_PROCESSORS()
        pid (i) = i
      END DO
      CALL  open (pid)
      ...

    EXTRINSIC (HPF_LOCAL) SUBROUTINE open (pid)
      INTEGER :: pid (:)
!HPF$ DISTRIBUTE pid (BLOCK)
      CHARACTER*7 :: filename

      WRITE (filename,'(A4,I3.3)') "tmp.", pid (1)
      OPEN (UNIT = 10, FILE = filename, FORM = "unformatted")
    END
```

Figure 9: Using processor-dependent filenames in parallel I/O

these functions may be used in specification expressions, e.g. to declare array dimensions. We have already given some examples of this use in previous sections.

HPF also defines a new computational intrinsic function, `ILEN`.

Incidentally, HPF-1 extended 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. This extension has now been included in Fortran 95.

10.2 HPF standard library

HPF defines a *standard library* of procedures in a module called `HPF_LIBRARY`. It contains:

- 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`, `GRADE_DOWN`, `SORT_UP` and `SORT_DOWN`.

11 HPF 2.0 Approved Extensions

This section briefly summarises the HPF 2.0 Approved Extensions. The reader is referred to [19] for full details.

11.1 Extensions for data mapping

The HPF-1 directives for dynamic realignment and redistribution, `REALIGN`, `REDISTRIBUTE` and `DYNAMIC`, have become Approved Extensions in HPF 2.0.

In addition, extensions are provided for mapping pointers and components of derived types, and for mapping data to subsets of processors. The following new directives are defined: `GEN_BLOCK` and `INDIRECT`, for expressing irregular data distributions; `RANGE`, for specifying the range of possible distributions that an array may have; and `SHADOW`, for specifying the size of the 'overlap' or 'halo' region that should surround the local segment of an array to store adjacent off-processor data.

11.2 Extensions for data and task parallelism

The `ON` directive specifies which processor or processors are to perform a computation. The `RESIDENT` directive, to be used in conjunction with the `ON` directive, asserts that all accesses to a specified object within the scope of the `ON` directive are local to the executing processor. The `TASK_REGION` directive specifies the concurrent execution of different blocks of code on disjoint processor subsets.

11.3 Extensions for asynchronous I/O

An additional I/O control parameter is defined for unformatted `READ` and `WRITE` statements, which instructs them to return before completing the I/O, i.e. makes them *non-blocking*. An accompanying new statement, `WAIT`, is used to wait for the I/O operation to finish. These extensions permit I/O to be overlapped with computation.

11.4 Extensions to intrinsic and library procedures

Some new inquiry routines are defined, and some inquiry routines defined in the HPF 2.0 base language are extended to allow for the extended mapping features listed in section 11.1. A generalisation of the Fortran `TRANSPOSE` intrinsic is also defined.

11.5 Approved Extensions for HPF extrinsics

The rules and semantics of the extrinsic models `'SERIAL'` and `'LOCAL'`, and the extrinsic kinds `HPF_SERIAL`, `HPF_LOCAL`, `F77_LOCAL` and `'C'`, are defined as part of the Approved Extensions, as described in section 9. A module called `HPF_LOCAL_LIBRARY` is defined for use in `HPF_LOCAL` procedures.

12 Further information

This section lists some World Wide Web sites where further information about HPF compilers and tools, tutorials, example codes, and HPF-related projects can be found. By the

nature of such surveys the information given here is incomplete and is only intended as a starting point for further investigation. We apologise for omissions.

In general, a good starting point for finding information on HPF is the homepage of the HPF Forum at:

`www.crpc.rice.edu/HPFF/` or
`www.vcpc.univie.ac.at/information/mirror/HPFF/`

This also has all versions of the HPF Language Specification, and details of how to subscribe to the HPFF email list.

Another good starting point for finding HPF information is the homepage of the HPF User Group ('HUG') at:

`www.vcpc.univie.ac.at/information/HUG/`

A HUG email list has also been established. To subscribe to it, send an email to: `majordomo@vcpc.univie.ac.at` with the message body: `subscribe hug-l`.

12.1 HPF compilers and tools

Surveys of HPF compilers and tools can be found at the following sites:

`www.ac.upc.es/HPFSurvey/`
: A survey of commercial and public domain HPF compilers, summarising the features they support, conducted by the European Centre for Parallelism of Barcelona and Queen's University Belfast.

`www.irisa.fr/pampa/HPF/survey.html`
: A survey of commercial and public domain HPF compilers and tools, conducted by the PAMPA project at IRISA.

`www.crpc.rice.edu/HPFF/hpfcompiler/` or
`www.vcpc.univie.ac.at/information/mirror/HPFF/hpfcompiler/`
: Information about commercial HPF compilers on the HPFF homepage.

Some currently available commercial HPF compilers are as follows:

Vendor	Product	URL
ACE	EXPERT HPF	`www.ace.nl/compilers.html`
APR	xHPF	`www.apri.com/`
Digital	DIGITAL Fortran	`www.digital.com/info/hpc/fortran/hpf.html`
EPC	HPF Mapper	`www.epc.co.uk/hpf.html`
IBM	xlhpf	`www.software.ibm.com/ad/fortran/xlhpf/`
NA Software	HPF Plus	`www.nasoftware.co.uk/`
PGI	pghpf	`www.pgroup.com/`
Pacific-Sierra	VAST-HPF	`www.psrv.com/vast/vasthpf.html`

There are also some freely available public domain HPF compilation systems. The following two in particular are quite mature and stable:

Name	Origin	URL
ADAPTOR	GMD-SCAI	www.gmd.de/SCAI/lab/adaptor/adaptor_home.html
SHPF	U. Southampton & VCPC	www.vcpc.univie.ac.at/information/software/shpf/

Both of these systems comprise an HPF to Fortran translator and a communications library, and can be installed on any system that has Fortran and C compilers (and a C++ compiler for SHPF) and a message-passing implementation such as MPI. The SHPF release package also contains an inter-procedural analysis tool for Fortran and HPF programs, called IDA.

The following are some research prototypes of HPF-related compilation systems:

Name	Origin	URL
Annai	CSCS / NEC	www.irisa.fr/pampa/HPF/annai.html
EPPP	CRIM Montreal	www.CRIM.CA/apar/Group/English/Projects/EPPP.html
Fortran D	Rice U.	www.cs.rice.edu/fortran-tools/DSystem/DSystem.html
FX Compiler	CMU	www.cs.cmu.edu/~fx/
HPF+	U. Vienna	www.par.univie.ac.at/hpf+/
HPFC	Ecole des Mines	www.cri.ensmp.fr/~coelho/hpfc.html
Pandore	IRISA	www.irisa.fr/pampa/PANDORE/pandore.html
PARADIGM	UIUC	www.crhc.uiuc.edu/Paradigm/
VFCS	U. Vienna	www.par.univie.ac.at/project/vfcs.html

12.2 HPF tutorials

HPF training courses and/or on-line tutorial material are available from the following:

Source	URL
EPCC	www.epcc.ed.ac.uk/epcc-tec/documents/
U. Liverpool	www.liv.ac.uk/HPC/HPFpage.html
U. Syracuse	www.npac.syr.edu/users/haupt/tutorial/tutorial.html
VCPC, U. Vienna	www.vcpc.univie.ac.at/activities/tutorials/HPF/
Digital	www.digital.com/info/hpc/f90/hpf_tutorial.ps

12.3 HPF applications

The following site is building a repository of sample HPF codes, including code fragments, kernels and full applications:

www.npac.syr.edu/hpfa/

In addition, sample HPF codes are included in the release packages for the public domain HPF compilers SHPF and ADAPTOR (see Section 12.1), and sample codes are also available from some of the other Web sites listed above.

12.4 HPF-related projects

A list of some HPF-related projects is available on the HPFF homepage at:

`www.crpc.rice.edu/HPFF/projects.html` or
`www.vcpc.univie.ac.at/information/mirror/HPFF/projects.html`

In addition, many HPF-related projects, e.g. projects developing HPF applications, tools, interfaces to standard libraries, etc, were presented at the First HPF User Group meeting which was held in Santa Fe in February 1997. The abstracts and some of the slides of these presentations are available at:

`www.lanl.gov/HPF/`

The following are some HPF-related projects funded by the European Union's ESPRIT programme:

ESPRIT project	URL
HPC Standards	`www.ccg.ecs.soton.ac.uk/Projects/hpc-stds/`
HPF+	`www.par.univie.ac.at/hpf+/`
PHAROS	`www.vcpc.univie.ac.at/activities/projects/PHAROS/`
PPPE	`www.vcpc.univie.ac.at/activities/projects/PPPE/`
PREPARE	`www.irisa.fr/pampa/PREPARE/prepare.html`

13 Conclusions

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 a good chance that it will become one of the foremost languages for scientific and engineering applications on such platforms.

Having said this, writing *efficient* HPF programs is not necessarily 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 needs a good understanding of the program and of the HPF execution model in order to map data effectively. In addition, 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, for example 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 improve code legibility as well as efficiency.

Acknowledgements

We thank Harald Ehold for providing much of the information given in Section 12.

References

[1] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C-W. Tseng and M-Y. Wu. 'Fortran D Language Specification'. Technical reports COMP TR90-141, Dept. of Comp. Sci, Rice Univ, Houston, TX, Dec. 1990, and SCCS-42c, Syracuse Center for Comp. Sci, Syracuse Univ, Syracuse, NY, April 1991.

[2] M-Y. Wu and G. Fox. 'Fortran 90D compiler for distributed memory MIMD parallel computers'. Technical report SCCS-88b, Syracuse Center for Comp. Sci, Syracuse Univ, Syracuse, NY, July 1991.

[3] B. Chapman, P. Mehrotra and H. Zima. 'Programming in Vienna Fortran', *Scientific Programming*, 1 (1), August 1992.

H. Zima, P. Brezany, B. Chapman, P. Mehrotra and A. Schwald. 'Vienna Fortran—a Language Specification'. ICASE Interim Report 21, ICASE NASA Langley Research Center, Hampton, Virginia 23665, March 1992.

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

[5] J. H. Merlin. "Techniques for the Automatic Parallelisation of Distributed Fortran 90". Technical report SNARC 92-02, Dept. of Electronics and Comp. Science, Univ. of Southampton, November 1991.

[6] F. André, J.-L. Pazat and H. Thomas, "PANDORE: a system to manage data distribution", *Int. Conf. on Supercomputing*, pp 380–388, June 1990.

[7] Digital Equipment Corporation. 'DECmpp 12000 Sx—High Performance Fortran Reference Manual'. Report [AA-PMAHC-TE], Digital Equipment Corporation, Maynard, Massachusetts. February 1993.

[8] D. M. Pase, T. MacDonald and A. Meltzer. 'MPP Fortran Programming Model'. Cray Research Inc, Eagan, Minnesota, August 26, 1992.

[9] J. Sanz et al. 'Data Parallel Fortran'. Technical report, IBM Almaden Research Center, March 1992.

[10] MasPar Computer Corporation. 'MasPar Fortran Reference Manual'. MasPar Computer Corporation, 749 North Mary Avenue, Sunnyvale, California. May 1991.

[11] Thinking Machines Corporation. 'CM Fortran Reference Manual'. Thinking Machines Corporation, Cambridge, Massachusetts, July 1991.

[12] High Performance Fortran Forum. 'High Performance Fortran Language Specification version 1.0'. *Sci. Prog.* special issue, 2 (1 and 2), 1993. Also available at:
`www.crpc.rice.edu/HPFF/hpf1/` or
`www.vcpc.univie.ac.at/information/mirror/HPFF/hpf1/`.

[13] High Performance Fortran Forum. 'High Performance Fortran Journal of Development'. *Sci. Prog.* special issue, 2 (1 and 2), 1993. Also available at:
`www.crpc.rice.edu/HPFF/hpf1/` or
`www.vcpc.univie.ac.at/information/mirror/HPFF/hpf1/`.

[14] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr and M.E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994. ISBN 0-262-61094-9. URL: `http://mitpress.mit.edu/book-home.tcl?isbn=0262610949`.

[15] J. H. Merlin and A. J. G. Hey. An introduction to High Performance Fortran. *Scientific Programming*, 4,87–113, 1995.

[16] High Performance Fortran Forum. 'High Performance Fortran Language Specification version 1.1'. URL `www.crpc.rice.edu/HPFF/hpf1/` or `www.vcpc.univie.ac.at/information/mirror/HPFF/hpf1/`.

[17] High Performance Fortran Forum. 'HPF-2 Scope of Activities and Motivating Applications'. November 1994.
URL `ftp://softlib.rice.edu/pub/HPF/hpf2-requirements.ps.gz`.

[18] ISO/IEC 1539-1:1997. Information Technology—Programming Languages—Fortran—Part 1: Base language

[19] High Performance Fortran Forum. 'High Performance Fortran Language Specification version 2.0'. URL `www.crpc.rice.edu/HPFF/hpf2/` or `www.vcpc.univie.ac.at/information/mirror/HPFF/hpf2/`.