Parallel Programming in *High Performance Fortran*

D.B. Carpenter

Computing Services,
Southampton University,
Southampton, S09 5NH

Introduction

For several years it has been widely acknowledged that the route to very high performance computing is through parallel processing. Today it seems that individual processors are approaching their technological limits, and the most straightforward way to surpass those limits is by allowing many processors to cooperate on a single task.

"Parallel computers" of one kind or another have been on the market for over a decade. The spectrum of machines is very broad. At one extreme there are tightly coupled systems involving thousands of very specialised "processing elements"—individually, hardly recognisable as computers. At the opposite extreme, local area networks of ordinary workstations can be programmed to cooperate on single tasks, and to this extent qualify as parallel computers. Somewhere in between are systems built from microprocessors specially designed to support parallel computing; and systems built around more or less conventional processors, with attached co-processors, switching circuits, shared memory subsystems, etc, to support very high bandwidth communication amongst the processors.

This divervity of architectures is a big headache for the programmer. Arguably, this is why parallel machines do not yet dominate the market for high performance computers. The problem is that a program written for one brand of parallel computer may not run at all on another manufacturer's machine. Even if you stick with the same manufacturer, their next generation machine may look completely different, and your code may have to changed to exploit the new hardware.

Eventually it may happen that one architecture emerges as the de facto standard, and all manufacturers produce variations on that theme. To some extent this has happened with sequential computers. To a larger extent, the portability of sequential code can be attributed to the fact that all manufacturers provide compilers for the standard high-level languages. These conceal differences between their processors at the instruction level (even for sequential computers assembly-language programs are not portable). It does not seem to be possible to take the standard sequential languages and compile them to execute efficiently on parallel computers. But the precedent does suggest that the way forward in parallel computing may be to develop standard *parallel* languages which can be compiled to run on a wide range of architectures.

This development started already with the establishment of the Fortran 90 standard. Fortran 90 is the successor to Fortran 77. From the parallel programming point of view, it is interesting because it includes a set of array operations which can be used to express parallel algorithms. Fortran 90 is still quite conservative so far as parallel extensions are concerned.

In the last year the *High Performance Fortran Forum*, a coalition of many leading industrial and academic groups in the field of parallel processing, has established an informal language standard called *High Performance Fortran*

(HPF). It is based on Fortran 90, but extends the set of parallel features, and provides extensive support for computation on *distributed memory* parallel computers. The standard is supported by a majority of vendors of parallel hardware, including Cray, DEC, Fujitsu, HP, IBM, Intel, Maspar, Meiko, nCube, Sun and Thinking Machines. Several companies have already announced their intention to develop HPF implementations, including hardware vendors DEC, Intel, Maspar, Meiko, Thinking Machines, and compiler vendors ACE, APR, KAI, Lahey, NA Software, Portland and PSR.

An official subset of HPF has been defined to facilitate early implementations. It is expected that the first true *Subset HPF* implementations will appear at the end of 1993 or start of 1994.

This document is an introduction to data parallel programming in Fortran 90 and High Performance Fortran. It is *not* an exhaustive or unbiased description of either of the two languages. It concentrates on a few interesting aspects of the languages, and illustrates how to program in them.

Part I concentrates on how parallelism is expressed in the languages. Part II describes the HPF model of distributed data.

Contents

Part I
Fortran 90 and data-parallel computation

Fortran 90, and especially its extension, High Performance Fortran, are designed to express of the kind of parallelism which occurs most frequently in numerical programs—*data-parallelism*.

In computations involving large arrays it often happens that processing can happen independently for each element of a data structure. Under favourable conditions, one can exploit a correspondingly large number processors to execute all the elemental computations in parallel. This provides scope for a potentially enormous increase in execution speed.

There are various obstacles to exploiting the data-parallelism inherent in many numerical algorithms. The first problem is to write the algorithm in a form which makes the potential parallelism explicit. The second, less obviously, is to somehow ensure that each processor has rapid access to the array elements required to complete its individual task. Otherwise the costs of moving data between many processors may outweigh the benefits of parallel execution.

The second task is probably the more painful one, and much of HPF is dedicated to providing the programmer with means of approaching it. The first part of this document, however, concentrates on the way in which parallelism itself is expressed in HPF.

As it turns out, the language extensions needed to express data-parallelism are quite modest. Essentially all that is needed is the facility to manipulate arrays-valued variables and expressions on a similar footing to scalars. This is as much as pure Fortran 90 provides to support parallelism. HPF goes a little further, and provides a `FORALL` construct, which is something like a parallel `DO` loop.

The first section here describes the new Fortran 90 array features. The second describes the HPF `FORALL` construct. Both sections include many illustrative code fragments. Section 3 contains some more complete examples of parallel codes.

Fortran 90 contains many new features which are not discussed here, including modules and derived types. These are extremely useful, but are not essentially connected with *parallel* programming. Also, the syntax of Fortran has been extended to allow more structured layout of programs. We assume that readers will not be too unnerved to see `ENDDO` appearing in the examples instead of `CONTINUE`, or statements starting before column 7! Fortran traditionalists (and people with old codes to port) can rest assured that all the idiosyncracies of earlier standards are retained in Fortran 90, alongside the more modern forms.

1 Data-parallel computation in Fortran 90

1.1 Some array terminology

In Fortran, any variable or expression is either *scalar*, or has a non-zero *rank*. In the second case we are dealing with an array; the "rank" is the number of dimensions to that array.

```
REAL A (5, 10), B (11:15, 10), C (11:15, 11:20)
```

declares `A`, `B` and `C` as rank-2 arrays (of real numbers). In Fortran the index of an array starts at 1 by default, but index ranges with other starting points can be specified. This has been done for the first dimension of `B` and both dimensions of `C`. Although the indices associated with the first dimension of `A` and `B` take different values (1 to 5 for `A`; 11 to 15 for `B`), the *extent* they range over is 5 in both cases. Similarly, the extent of the second dimension is 10 for all three arrays.

The vector of extents associated with an array is called the *shape* of the array. `A`, `B` and `C` all have shape $(5, 10)$. Two arrays are *conformable* if they have the same shape. So `A`, `B` and `C` are certainly conformable.

1.2 Fortran 90 array expressions

The most primitive kind of array expression is the "array constructor". This is a list of items in special brackets. For example

```
(/ 2, 3, 5, 7 /)
```

is a constant array of shape (4). We won't have a great deal to say about array constructors, but two things are worth mentioning: lists inside array constructor brackets can include implied do loops; and array constructors only yield one-dimensional arrays, but the intrinsic function `RESHAPE` can be used to obtain higher rank array values.

Turning to arithmetic, any of the built-in operations of Fortran 90 can be applied directly to conforming arrays. If `A`, `B` and `C` are declared as in the previous section,

```
A + B
B * C
A + (B * C)
```

are legal expressions. Their results are arrays of the same shape as the operands. `B * C` stands for the table of values

$$\begin{array}{ccc} B(11,1) * C(11,11) & \cdots & B(11,10) * C(11,20) \\ \vdots & \ddots & \vdots \\ B(15,1) * C(15,11) & \cdots & B(15,10) * C(15,20) \end{array}$$

Values held in corresponding positions in the arrays are multiplied together. In general, as here, corresponding "positions" may be associated with different indices, if the index ranges are declared differently.

As a special case, scalars are "conformable" with any array. If `D` is scalar

```
REAL D
```

then `C + D` represents

$$\begin{array}{ccc} C(11,11) + D & \cdots & C(11,20) + D \\ \vdots & \ddots & \vdots \\ C(15,11) + D & \cdots & C(15,20) + D \end{array}$$

In addition to arithmetic operations, many of the Fortran intrinsic functions can be applied "elementally". Hence `COS (A)` means the array of values

$$\begin{array}{ccc} COS(A(1,1)) & \cdots & COS(A(1,10)) \\ \vdots & \ddots & \vdots \\ COS(A(5,1)) & \cdots & COS(A(5,10)) \end{array}$$

This possibility doesn't extend to user-defined functions in Fortran 90, but see section 2.2.

Array expressions (and variables) can also be formed by taking a *sections* of a named array object. An array section is built from some subset of the elements of an array object—those associated with a selected subset of the index range attached to the object. (Incidentally, the restriction to "named object" is significant. In Fortran index ranges are associated with declared, named arrays. It isn't possible to take an element or section of an unnamed expression, even if it is array-valued.)

The simplest array section is formed by subscripting an array by colon-separated range of index values. Suppose an array `X` is declared by

```
INTEGER X (0:12)
```

Then `X (1:3)` is a section containing the second, third and fourth elements of `X` (starting at the second element of `X`, because indexing of `X` starts at 0). It is an array-valued expression of shape (3), and can be used in the same context as any other such expression. Either the lower bound or the upper bound can be omitted in the index range of a section, in which case they default to the lowest or highest values taken by the array's index. So `X (:)` is a section containing the whole of `X`.

This simple form of array section can be generalised with a "stride". `X (1:10:3)` is an array section of size 4 containing every third element of `X` with indices between 1 and 10 (ie, indices 1, 4, 7, 10). Even more generally, if `Y` is a one dimensional integer array, `X (Y)` is an array section of the same length as `Y`, built from elements of `X` by taking each element of `Y` in turn as an index into `X`.

For multi-dimensional arrays, some dimensions could be subscripted with a normal scalar expression, and some could be "sectioned". The rank of the resulting array variable is the number of dimensions that are sectioned.

1.3 Array assignments

Naturally, we can use array-valued expressions in assignments. The simplest version has exactly the same syntax as the scalar Fortran assignment. The expression on the right hand side of the assignment must be conformable with the array variable on the left hand side.

As a first, elementary example of a "data parallel" program, let's take

```
REAL HORIZ (10), VERT (10), HYPOT (10)

HYPOT = SQRT (HORIZ**2 + VERT**2)
```

Assume HORIZ and VERT were initialised to the lengths of the horizontal and vertical sides of ten right-angled triangles. The result, returned in the array HYPOT is the length of the ten hypotenuses[1].

The effect of an array assignment is that first all elements of the expression on the right-hand side are evaluated, *then* they are assigned to the elements of the variable on the left-hand-side. This statement is than less vacuous than it sounds. For example the statement

```
X (0:9) = X (-1:8)
```

[1] We need to check that the expression on the right hand side of the assignment is a legal array expression, according to the rules given above. HORIZ is an array of shape (10). The constant 2 is a scalar, conformable with any array, and `**` is a built-in arithmetic operator which can be used to combine any conformable expressions. So the subexpression `HORIZ**2` is a legal array expression of shape (10). The same goes for the sub-expression `VERT**2`, and the sum of these two expressions. Finally `SQRT` is an "elemental" intrinsic function. The full expression is conformable with the array variable on the left hand side, so we have a legal program.

replaces all elements of `X` with indices between 0 and 9 by the values of the preceding elements. It has a different effect from the loop

```
DO i = 0, 9
  X (I) = X (I - 1)
ENDDO
```

which replaces all the elements of `X` with indices between 0 and 9 with the value of `X (-1)`.

Returning to parallel programming with a less trivial problem, let's try to sum the elements of an array, using a parallel algorithm.

```
INTEGER INPUT (16)
INTEGER SUM, I

SUM = 0
DO I = 1, 16
  SUM = SUM + INPUT (I)
ENDDO
```

is a sequential program which achieves the required result. It takes 16 sequential steps to get there. A more parallel approach is to sum the 16 numbers pairwise in parallel, to give 8 results. These 8 are then summed pairwise in parallel to give 4 numbers, and the process is repeated until a single result is obtained. This can be written as follows

```
INTEGER INPUT (16), TEMP1 (8), TEMP2 (4), TEMP3 (2)
INTEGER SUM

TEMP1 = INPUT (1:8) + INPUT (9:16)
TEMP2 = TEMP1 (1:4) + TEMP1 (5:8)
TEMP3 = TEMP2 (1:2) + TEMP2 (3:4)
SUM   = TEMP3 (1) + TEMP3 (2)
```

Four parallel steps reduce the set to a single result. In principle this could save a factor of $16/4 = 4$ in execution time. The program can be generalised with a loop

```
INTEGER INPUT (16), TEMP (16)
INTEGER SUM, I

TEMP = INPUT
I    = SIZE (INPUT)
DO WHILE I > 1
  J = I / 2
  TEMP (1 : J) = TEMP (1 : J) + TEMP (J + 1 : I)
```

```
  I = J
ENDDO
SUM  = TEMP (1)
```

(`SIZE` is a new intrinsic function in Fortran 90, which can be used to find the extent of an array). This will work for any `INPUT` array whose size is a power of two. It gives a result in $\log_2$(`SIZE (INPUT)`) iterations.

A second form of array assignment allows one to restrict the assignment to some subset of the array variable on the left-hand-side by specifying a "mask". A mask is `LOGICAL` array expression, conformable with the variable. Assignment of the corresponding element of the array expression on the right-hand is only performed where the mask takes value `.TRUE.`. Where the mask is `.FALSE.`, the variable is left unchanged. For example, suppose `EVEN` is a logical array with indices from 0 to 9, whose even-indexed elements are `.TRUE.` and odd-indexed elements are `.FALSE.`. Then

```
WHERE EVEN
  X (0:9) = X (1:10)
ENDWHERE
```

replaces the values in the section `X(0:9)` having even indices with copies of the values of the next element in the array. `WHERE` constructs can have optional `ELSEWHERE` clauses, which look rather similar to `ELSE` clauses in `IF` constructs. This is often useful. It can have slightly surprising effects if variables assigned in the main clause are referenced in the `ELSEWHERE` clause. Take the code

```
WHERE EVEN
  X (0:9) = X (1:10)
ELSEWHERE
  X (0:9) = X (-1:8)
ENDWHERE
```

We might hope that the effect is to replace each even-indexed element of `X` with the next element of the array, and replace each odd-indexed element with the previous element, ie to swap adjacent odd and even elements, $X(0) \leftrightarrow X(1)$, $X(2) \leftrightarrow X(3)$, etc. *In fact*, the `WHERE/ELSWHERE` construct is defined to make the above program equivalent to

```
WHERE EVEN
  X (0:9) = X (1:10)
ENDWHERE
WHERE .NOT. EVEN
  X (0:9) = X (-1:8)
ENDWHERE
```

which has a different effect—the first assignment copies the following odd element down into each even element, but the second assignment copies the *new* value back to where it originally came from. So the second assignment effectively does nothing.

In general both the WHERE and ELSEWHERE clauses can contain any number of assignments, so long as all of the array variables are conformable. The meaning is defined by splitting the construct into a series of consecutive single-assignment WHERE constructs, as above.

1.4 Arrays as arguments and results of subprograms

Consider the following subprogram definition

```
FUNCTION MATPROD (A, B)
REAL A (:, :), B (:, :)
REAL MATPROD (SIZE (A, 1), SIZE (B, 2))
  DO I = 1, SIZE (A, 1)
    DO J = 1, SIZE (B, 2)
      MATPROD (I, J) = DOT_PRODUCT (A (I, :), B (:, J))
    ENDDO
  ENDDO
ENDFUNCTION
```

This example introduces several new features of Fortran 90. Firstly, an *assumed shape* dummy argument is an array whose index (upper) bounds are undefined, as for A and B (a lower bound can be optionally specified, but it defaults to 1, as here). The upper bounds are set according to the shape of the actual argument. Secondly SIZE has an optional argument that defines which dimension of a multi-dimensional argument is required, so SIZE (B, 2) is the extent of the second dimension of B. Thirdly DOT_PRODUCT is a new intrinsic function which returns the scalar product of two vectors.

Last but not least, the result of a function can be an array. Furthermore, the size of this array can depend on the arguments, as in the example.

In various circumstances, including the case when a dummy argument is assumed size, and when a function returns an array result, the called procedure must have an *explicit interface* in the calling program. There are various ways to make a subprogram's interface explicit. The simplest is to include an *interface block* amongst the declarations in the calling program. A program calling MATPROD might contain the declaration

```
INTERFACE
  FUNCTION MATPROD (A, B)
  REAL A (:, :), B (:, :)
  REAL MATPROD (SIZE (A, 1), SIZE (B, 2))
  ENDFUNCTION
ENDINTERFACE
```

This contains just the information (the attributes of the arguments and result) needed for the compiler to generate the correct calling code. Typically, one might collect groups of interfaces in text files, and INCLUDE such files in programs calling procedures from a group. In full Fortran 90, the module system incorporates a more controlled way of making interfaces explicit.

1.5 Fortran 90 array intrinsics

The set of Fortran intrinisic functions has been extended to include many array operations. One of these functions (DOT_PRODUCT) was introduced in the previous section.

Some other notable inclusions are the "array reduction" functions including SUM and PRODUCT which add or multiply together elements of an array; MAXVAL and MINVAL which return largest or smallest elements of an array; and ANY and ALL, which take the disjunction and conjunction of the elements of a LOGICAL array.

Other functions include the SPREAD function which "adds an extra dimension" to an array, returning an array in which values from the original array are replicated over the range of the new index; TRANSPOSE, which returns an array containing in the elements of its two dimensional argument, transposed; and shift operations which return arrays with values shifted by a constant index offset in one of the dimensions of the argument array.

We can assume (or hope) that these functions have been implemented using the best available—where appropriate, parallel—algorithms. So, for example SUM may have been implemented along the lines described in section 1.3.

Creative use of the array manipulation functions greatly extends the class of parallel algorithms we can express in Fortran 90. But in many cases calls to functions like SPREAD, TRANSPOSE and the shifts can be eliminated, and readability of programs improved, by using the HPF construct FORALL.

2 HPF extensions

2.1 FORALL assignments

Together, the features of Fortran 90 described so far provide quite a powerful framework for implementing data parallel algorithms, as the examples in section 3 will illustrate.

High Performance Fortran extends the set of parallel statements in one way—it includes a FORALL construct. FORALL has some similarities to the conventional DO loop; in some other respects it resembles the Fortran 90 WHERE construct.

```
FORALL (I = 0 : 9)
  X (I)  = X (IEOR (1, I))
ENDFORALL
```

is one way of achieving the exchange of adjacent odd and even elements[2], which we failed to effect straightforwardly with a `WHERE` construct at the end of section 1.3. When a `FORALL` contains a single assignment, as here, all expressions on the right-hand sides are evaluated before any element on the left-hand side is assigned. In this respect a single-assignment `FORALL` is akin to a simple array assignment, rather than the `DO` loop it superficially resembles.

The body of a `FORALL` may contain several statements, but these are essentially restricted to assignments or nested `FORALL` constructs, and as for a `WHERE` construct the meaning is as if there were several consecutive `FORALL`s, each containing a single statement. Something like

```
FORALL (I = 0 : 9 : 2)
  X (I)     = X (I + 1)
  X (I + 1) = X (I)
ENDFORALL
```

(note the option of a stride in the `FORALL` index range) is defined to be equivalent to

```
FORALL (I = 0 : 9 : 2)
  X (I)     = X (I + 1)
ENDFORALL
FORALL (I = 0 : 9 : 2)
  X (I + 1) = X (I)
ENDFORALL
```

so it has a different effect from the first example in this subsection, and the same effect as the earlier `WHERE/ELSEWHERE` example.

A single `FORALL` can define several indices. So

```
FORALL (I = 1 : SIZE (A, 1), J = 1 : SIZE (B, 2))
  MATPROD (I, J) = DOT_PRODUCT (A (I, :), B (:, J))
ENDFORALL
```

is a completely parallel version of the body of `MATPROD`, introduced in section 1.4.

2.2 Pure procedures

The order of execution of the individual assignments in a `FORALL` statement is undefined—ideally they all execute in parallel. Suppose arbitrary expressions could appear inside the body of a `FORALL`. Then in particular calls to arbitrary user-defined functions could appear. A user-defined function may have a variety

[2] `IEOR` is "bitwise exclusive or"—combining a number with 1 using this function inverts the least significant bit: `IEOR (1, 0) = 1`, `IEOR (1, 1) = 0`, `IEOR (1, 2) = 3`, `IEOR (1, 3) = 2`, etc.

of *side-effects*. It might, for example, modify a variable in a common block, or perform some read or write operation.

Whenever side-effects like this can occur, it's preferable that they should happen in a well-defined order, otherwise the net behaviour of the program may be non-deterministic (to belabour the point, if one statement modifies a variable which another reads, the overall behaviour of the program depends on the order in which those statements were executed). Some parallel languages accept the possibility of non-deterministic programs. Fortran isn't one of them.

Rather than completely forbid reference to user-defined functions inside `FORALL` constructs, a special class of side-effect free functions has been defined, which can safely be called in parallel. A *pure* procedure (function or subroutine) is defined with the word `PURE` at the front of its header line:

```
PURE FUNCTION PYTHAG (X, Y)
REAL, INTENT (IN) :: X, Y
REAL PYTHAG
  PYTHAG = SQRT (X**2 + Y**2)
ENDFUNCTION
```

and is then forbidden from modifying global data, or performing any IO operation, or calling any non-pure procedure[3].

A function like `PYTHAG` may now appear in the body of a `FORALL`

```
FORALL (I = 1 : 10)
  HYPOT (I) = PYTHAG (HORIZ (I), VERT (I))
ENDFORALL
```

(It would be pleasant if `PYTHAG` could also be applied "elementally" to arrays, as in

```
HYPOT (1 : 10) = PYTHAG (HORIZ (1 : 10), VERT (1 : 10))
```

which would achieve the same effect as the `FORALL`. This possibility has been omitted from the HPF language definition, purely to minimise points of departure of HPF from the existing Fortran 90 standard.)

3 Some example parallel programs

We conclude this discussion of data parallel programming in Fortran by displaying a couple of more substantial applications. These have been chosen to

[3]Aside 1: In Fortran 90 it is possible to specify the *intent* of a dummy argument. If an argument has the `INTENT (IN)` attribute, the programmer guarantees not to change the value of the argument in the body of the procedure. Rather pedantically, HPF insists that `PURE` functions have this attribute explicitly specified for all dummy arguments.

Aside 2: The double colon form of declaration can be used to specify more than one attribute—always including the type as the first—in a single declaration. In general a comma-separated list of attributes is separated from the list of variables by `::`

illustrate the parallel features of the Fortran 90. The first is a parallel implementation of a standard matrix algorithm. The other two are implementations of more intricate parallel algorithms. Explanation of the working of these algorithms is beyond the scope of this report.

They are coded in pure Fortran 90, without HPF extensions. They have been tested using an existing Fortran 90 compiler.

3.1 Linear solver

Figures 1 and 2 give a program for solving of linear equations by the LU-decompostion technique. It illustrates data-parallel programming through Fortran 90 array syntax. The implementation of `EXCHANGE`, which swaps the contents of its two arguments, is omitted.

Section 5.4 takes a simplified kernel of this program as a basis for a distributed data programming example.

Note the use of the `SPREAD` intrinsic for converting one-dimensional arrays into two-dimensional ones. This is a common feature of programs written using pure Fortran 90 array syntax. The version given in section 5.4 uses an HPF `FORALL` construct, and is rather more transparent.

The number of parallel operations needed to complete the program is proportional to the matrix size, N (whereas the total number of arithmetic operations is proportional to N^3).

```
SUBROUTINE LINEAR_SOLVER (A, B)
REAL, DIMENSION ( : , : ) :: A, B
! solve A X = B.  results for X overwrite B.  A is also overwritten.
  INCLUDE 'interfaces.inc'

  CALL LU_DECOMPOSE (A, B)
  CALL SOLVE_LOWER  (A, B)
  CALL SOLVE_UPPER  (A, B)
ENDSUBROUTINE

SUBROUTINE LU_DECOMPOSE (A, B)
REAL, DIMENSION ( : , : ) :: A, B
! Solve L U = A.  Results for L and U overwrite A.  B is
! pivotted as A.
  INTEGER N, R, R1, PIVOT (1)
  REAL, DIMENSION (SIZE (A, 1)) :: L_COL, U_ROW

  N = SIZE (A, 1)
  DO R = 1, N - 1
    PIVOT = R - 1 + MAXLOC (ABS (A (R : , R)))
    CALL EXCHANGE (A (R, : ), A (PIVOT (1), : ))
    CALL EXCHANGE (B (R, : ), B (PIVOT (1), : ))

    R1 = R + 1
    L_COL (R  : ) = A (R : , R)
    U_ROW (R1 : ) = A (R, R1 : ) / L_COL (R)
    A (R    , R1 : ) = U_ROW (R1 : )
    A (R1 : , R1 : ) = A (R1 : , R1 : ) - &
      SPREAD (L_COL (R1 : ), 2, N - R) * SPREAD (U_ROW (R1 : ), 1, N - R)
  ENDDO
ENDSUBROUTINE
```

Figure 1: Linear equation solution by LU-decomposition

```
SUBROUTINE SOLVE_LOWER (L, B)
REAL, DIMENSION ( : , : ) :: L, B
! Solve L Y = B.  Result for Y overwrites B.
  INTEGER N, M, R, R1
  REAL L_COL (SIZE (L, 1)), Y_ROW (SIZE (B, 2))

  N = SIZE (L, 1)
  M = SIZE (B, 2)
  DO R = 1, N
    R1 = R + 1
    L_COL (R : ) = L (R : , R)
    Y_ROW        = B (R, : ) / L_COL (R)
    B (R    , : ) = Y_ROW
    B (R1 : , : ) = B (R1 : , : ) - &
      SPREAD (L_COL (R1 : ), 2, M) * SPREAD (Y_ROW, 1, N - R)
  ENDDO
ENDSUBROUTINE

SUBROUTINE SOLVE_UPPER (U, Y)
REAL, DIMENSION ( : , : ) :: U, Y
! Solve U X = Y.  Result for X overwrites Y.
  INTEGER N, M, R, R1
  REAL U_COL (SIZE (U, 1)), X_ROW (SIZE (Y, 2))

  N = SIZE (U, 1)
  M = SIZE (Y, 2)
  DO R = N - 1, 1, -1
    R1 = R + 1
    U_COL ( : R) = U ( : R, R1)
    X_ROW        = Y (R1, : )
    Y ( : R, : ) = Y ( : R, : ) - &
      SPREAD (U_COL (: R), 2, M) * SPREAD (X_ROW, 1, R)
  ENDDO
ENDSUBROUTINE
```

Figure 2: Linear equation solution (details)

3.2 Parallel sorting

Parallel sorting is a more intricate problem. The previous example took a standard sequential algorithm and parallelised it. Here, we need a specialised parallel algorithm. The one used here is odd-even merge sort. Parallel merge sorts work on a "divide and conquer" basis. The data is split into two halves which are individually sorted (in parallel). Then the two sorted halves are merged by a procedure which gives a sorted whole. This strategy is applied recursively, to sort the data set in $log_2(N)$ stages, provided the array size N is a power of two. The divide and conquer strategy is handled by the recursive function `SORT`. Merging is handled by `OE_MERGE` (which also happens to be a recursive procedure). Detailed discussion of the odd-even merge algorithm is omitted here.

The program illustrates the use of recursive procedures in Fortran 90. An explicit interface of `OE_MERGE` is needed (in `SORT`) because it takes assumed shape array arguments—this would be included in the file "interface.inc". It also illustrates the use of general regular array sections, with strides.

The version given in figure 3 is not fully parallelised. That requires the two invocations of `SORT` appearing as arguments to `OE_MERGE` to be evaluated in parallel, and that the two recursive invocations of `OE_MERGE` (in the assignments to `SOD` and `SEV`) are evaluated in parallel. Making the parallelism in these operations completely explicit, simply using array syntax, would involve a certain amount of code reorganisation. Neither Fortran 90 nor full HPF provide very good support for the divide and conquer paradigm[4].

A fully parallel implementation would involve a number of parallel steps propertional to $(log_2(N))^2$.

3.3 Parallel list ranking

The list ranking problem: the entries in a linked list is of N items are held in an array of size N. The array `NEXT` holds the links; the entry `NEXT (I)` is the index of the successor of element `I` in the list. The end of the list is flagged by the condition `NEXT (I) = I`. The problem is to find the distance of each element from the end of the list.

Figure 4 is an implementation of a standard parallel algorithm for solving this problem. It illustrates the use of "irregular sections"—array sections with vector subscripts.

The program completes in a number of parallel steps proportional to $log_2(N)$.

[4] In full HPF one could combine pairs of arrays into single arrays, declare the functions `PURE`, and use `FORALL` constructs with an index running over two values to achieve the extra parallelism. This is slightly obscure, and it's debatable whether such convoluted code would compile efficiently.

```
RECURSIVE FUNCTION SORT (S) RESULT (OUT)
INTEGER, DIMENSION (:) :: S
INTEGER, DIMENSION (SIZE (S)) :: OUT

  INCLUDE 'interfaces.inc'
  INTEGER N

  N = SIZE (S)
  IF (N == 1) THEN
    OUT = S
  ELSE
    OUT = OE_MERGE (SORT (S (1 : N / 2)), SORT (S (N / 2 + 1 : N)))
  ENDIF
ENDFUNCTION


RECURSIVE FUNCTION OE_MERGE (S1, S2) RESULT (OUT)
INTEGER, DIMENSION (:) :: S1
INTEGER, DIMENSION (SIZE (S1)) :: S2
INTEGER, DIMENSION (2 * SIZE (S1)) :: OUT

  INTEGER, DIMENSION (SIZE (S1)) :: SOD, SEV
  INTEGER N

  N = SIZE (S1)
  IF (N == 1) THEN
    IF (S1 (1) < S2 (1)) THEN
      OUT = (/ S1 (1), S2 (1) /)
    ELSE
      OUT = (/ S2 (1), S1 (1) /)
    ENDIF
  ELSE
    SOD = OE_MERGE (S1 (1 : N - 1 : 2), S2 (1 : N - 1 : 2))
    SEV = OE_MERGE (S1 (2 : N     : 2), S2 (2 : N     : 2))
    OUT (1) = SOD (1)
    WHERE (SOD (2 : N) < SEV (1 : N - 1))
      OUT (2 : 2 * N - 2 : 2) = SOD (2 : N)
      OUT (3 : 2 * N - 1 : 2) = SEV (1 : N - 1)
    ELSEWHERE
      OUT (2 : 2 * N - 2 : 2) = SEV (1 : N - 1)
      OUT (3 : 2 * N - 1 : 2) = SOD (2 : N)
    ENDWHERE
    OUT (2 * N) = SEV (N)
  ENDIF
ENDFUNCTION
```

Figure 3: Odd-even Merge Sort

```
FUNCTION RANK (NEXT)
INTEGER NEXT (:)
INTEGER RANK (SIZE (NEXT))

  INTEGER, DIMENSION (SIZE (NEXT)) :: P, DISTANCE, INDEX

  N = SIZE (NEXT)
  INDEX = (/ (I, I = 1, N) /)
  P = NEXT
  WHERE (P /= INDEX)
    DISTANCE = 1
  ELSEWHERE
    DISTANCE = 0
  ENDWHERE
  I = 1
  DO WHILE (I < N)
    WHERE (P /= P (P))
      DISTANCE = DISTANCE + DISTANCE (P)
      P = P (P)
    ENDWHERE
    I = 2 * I
  ENDDO
  RANK = DISTANCE
ENDFUNCTION
```

Figure 4: List ranking by the “Doubling” technique

Part II

Distributed data in High Performance Fortran

The approach to parallel programming described in part I using the array syntax of Fortran 90 and HPF, is simple and surprisingly general. In practise, though, it is usually too difficult for a compiler to take an arbitrary program written in this style and automatically execute it *efficiently* in parallel. To achieve that goal, the compiler will typically need more detailed information from the programmer about how the problem is to be divided across the parallel computer. Most of the HPF extensions to Fortran address the issue of providing the programmer with means to express this information.

The second part of this document describes these extensions. Once again, we will be selective in which parts of the language are presented here. Notably, we will not describe dynamic data mappings, or the new intrinsic or library functions.

The next section discusses in general terms the problems of programming distributed memory computers. This is followed by a section describing the HPF data mapping directives. For illustration, it includes a distributed version of the LU decomposition example given in part I. The last section describes the HPF subprogram interface (passing parameters to procedures is a delicate issue in the presence of distributed data).

4 Multi-processing and distributed data

The most successful parallel computers presently available are built from a number of largely autonomous processors, each possessing its own local memory area (figure 5). A processor can access values in its local memory very rapidly. It can also access values in another processor's memory, either by special machine instructions, or through message-passing software. But with current technology remote memory accesses are much slower (often by orders of magnitude) than local ones. This puts an onus on the programmer or compiler to minimise the number of accesses any processor makes to non-local memory. This is the *communication* problem.

A second problem is ensuring that each processor has a fair share of the total work load. Obviously, if most work ends up being done by one processor the benefits of parallelism are lost. This is the *load-balancing* problem.

High Performance Fortran allows the programmer to add various *directives* to a program. These tell the compiler how program data is to be distributed amongst the memory areas associated with a set of (real or virtual) processors. They *don't* allow the programmer to state directly which processor will perform

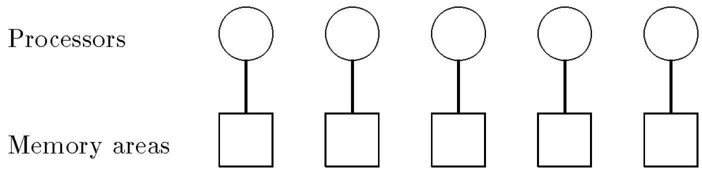

Figure 5: Idealised picture of a distributed memory parallel computer

a particular computation. But it is expected that if the operands of a particular sub-computation (for example, an assignment) are all found on the same processor, the compiler will allocate that part of the computation to the processor holding the operands, whereupon no remote memory accesses will be involved.

Hence the main burden on the HPF programmer is to distribute the program's data across the processors in such a way that

- as many sub-calculations as possible involve groups of operands allocated to the same processor, to minimize remote memory accesses, while
- the set of *independent* sub-calculations which can proceed in parallel at any time should involve data on as many different processors as possible, to maximise parallelism.

These may sound like conflicting goals. Sometimes they are, and programs which look highly parallel are nevertheless difficult to distribute effectively. Luckily, equally often it is possible to perform the juggling act successfully.

To make these points a bit more concrete, imagine an in idealised situation where the a particular step in a program contains p sub-calculations which can proceed in parallel. They might be the p elemental assignments making up an array assignment or `FORALL` construct. For the sake of argument, suppose each expression to be computed involves two operands. Together with the variable being assigned, this makes three operands for the whole sub-calculation. Also, suppose there are p processors available. (In general, the numbers of sub-calculations and processors are likely to be different—this case is just illustrative).

Figure 6 illustrates the situation where all operands of each sub-calculation reside in different memory areas. This means that each assignment involves at least two communications, wherever the calculation is performed.

Figure 7 illustrates a situation in which no communication is *necessary*, because all operands of *all* the assignments live on the same processor. But in this case the calculation will generally be performed where the data is, so no effective parallelism occurs. (Alternatively, the compiler might decide to share the tasks out anyway. But then *all* operands of tasks on processors other than the first would have to be communicated.)

Figure 8 illustrates the ideal situation, where all operands of an individual assignment occur on the same processor, but the operand groups of the sub-

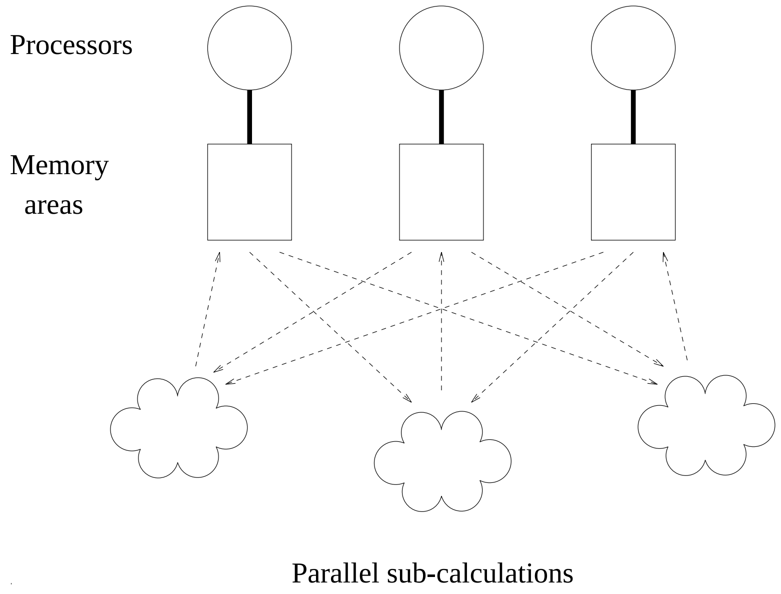

Figure 6: A data distribution leading to excessive communication

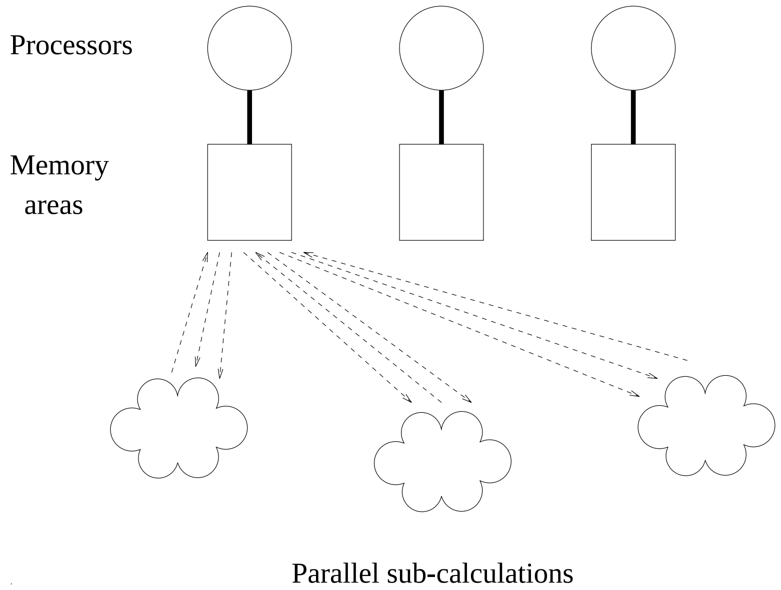

Figure 7: A data distribution leading to poor load balancing

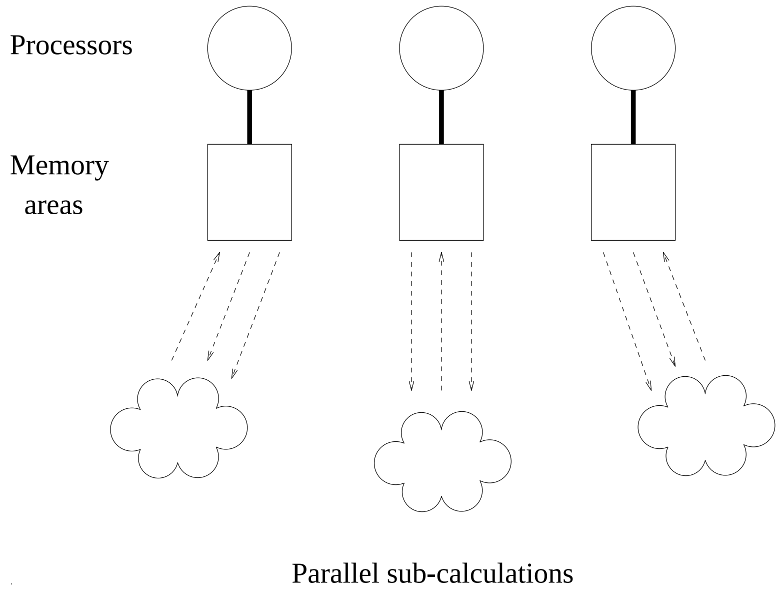

Figure 8: An ideal data distribution

calculations are uniformly distributed over processors. In this case we can rely on the compiler to allocate each computation to the processor holding the operands, requiring no communication, but perfect distribution of workload.

Except in the most trivial programs, it will never be possible to choose a distribution of program variables over processors so that *all* steps in the calculation look like figure 8. The art of distributed memory programming is to arrange things so that the most critical regions of the program look as much like this as possible.

There was a certain awkwardness to the descriptions of these three situations. In the last case one could generally rely on the compiler to allocate sub-calculations to processors in the obvious way. In the first two cases one has to hedge one's bets, and say that compiler might allocate computations over processors in various ways, but in any case the outcome is likely to be unfavourable. Given that HPF places an onus on the programmer to think about these kinds of issues, one might feel it would have been appropriate to allow the programmer fuller control over where particular sub-computations were performed (rather than have the programmer second-guess the compiler). Well, this argument has some force, but it isn't how HPF has been defined. As already emphasized, the programmer can define where particular data items are stored, but not where calculations are performed.

5 Data mapping directives

5.1 HPF directives

For the purposes of this document, HPF directives will be written *as if* they were part of the Fortran language. Please note well that in fact the standard defines these directives to be "structured comments". All the new "statements" introduced in the remainder of this document would have to be preceded with the text "`!HPF$`" in real programs. So, looking ahead to the next section, when we write

```
PROCESSORS P (10)
```

the actual program text must be

```
!HPF$ PROCESSORS P (10)
```

There are two main arguments for making HPF directives into Fortran comments. Firstly, used correctly, they do not change the actual results of a program. They only effect its *performance* on a parallel machine. In so far as they have no effect on results, they have the same status as comments. Secondly, more pragmatically, an HPF program can be compiled by an ordinary Fortran 90 compiler, which will just ignore the HPF comments. Hence an HPF program can be tested and debugged on any sequential computer for which Fortran 90 is available. This is a credible argument, undermined slightly by the fact that the HPF extensions described in part I (`FORALL` and `PURE`) do *not* have the status of comments, and would not be accepted by an ordinary Fortran compiler.

In any case the justification for not writing HPF directives as comments in this document is aesthetic—the proliferation of "`!HPF$`" strings would clutter the examples.

5.2 The processor arrangement

The programmer is expected to specify how program variables are distributed between the memory areas associated with a set of processors. A necessary (or at least, desirable) prerequisite is for a program to be able define some representation of the set of processors it uses. This can be done through the `PROCESSORS` directive.

Before introducing the directive, we should say something about the relationship between physical and abstract processors. The parallel computer on which a program is run will consist of some fixed set of physical processors. By and large, when we talk of "processors" below we won't be referring to these processors, but to a set of logical or *abstract* processors, which may be larger or smaller in number than the set of processors in the machine. If there are fewer abstract processors than physical processors, probably some of the physical processors will be idle, or occupied with other programs. If there are more abstract processors than physical processors, the system will somehow simulate the number of processors specified. In any case, the programmer has the option of not worrying about the size of the target machine[5]. An abstract processor has an associated local memory area, just like a real processor.

The syntax of a `PROCESSORS` definition is similar to the syntax of an array definition in Fortran

```
PROCESSORS P (10)
```

introduces a set of 10 abstract processors, assigning them the collective name `P`.

Like ordinary arrays, abstract processor sets, or *processor arrangements*, can be multi-dimensional. For example

```
PROCESSORS Q (4, 4)
```

introduces 16 abstract processors in a 4 by 4 array. There are a couple of reasons why one might want to declare a processor arrangement to be multi-dimensional. Most importantly, the major data-structures in a program may be multi-dimensional arrays, and it may be that the best way to divide up this data over processors is by sub-dividing it into blocks of the same dimension as the whole arrays. In this situation it certainly simplifies matters if we can view the processor arrangement itself as an array of the relevant dimension. As a secondary consideration, some parallel computers are physically arranged as multi-dimensional grids, in such a way that nearest-neighbour communications are cheaper than more remote communications. By exploiting such knowledge, and mimicking the physical grid arrangement in the `PROCESSORS` declaration, a sufficiently determined programmer may be able to tweak performance[6].

An important point to appreciate is that a single program may contain many `PROCESSORS` declarations. There is no defined relation between two processor arrangements of different shapes[7]. Typically they will be independently mapped onto the same set of underlying physical processors.

5.3 The template abstraction, and distribution

Having seen how to define one or more target processor arrangements, we need to introduce mechanisms for distributing data arrays over those arrangements. Although HPF does allow arrays to be distributed over processors directly (see

[5] Strictly speaking, an HPF implementation isn't *required* to support sets of abstract processors whose number differs from the set of physical processors, but the better systems presumably will. There may be performance advantages in making the abstract processor set conform with the physical processor set

[6] Such machine-specific optimisations are slightly at odds with the philosophy of portable parallel programming. Also it seems likely that the physical processor arrangement will have less influence on communication speeds in forthcoming parallel machines.

[7] There is however, an implicit equivalence between processor arrangements of the *same* shape. See section 6.2

section 5.5) it is often more satisfactory to go through the intermediary of an explicit *template*.

We have already introduced the idea of a "processor grid" as a `PROCESSORS` array, and suggested that the *dimension* of that grid may well reflect the dimension of the principal data arrays in the program. Generally, though, it would be inappropriate for the *size* parameters of the abstract processor array to correspond to those of the data arrays of the algorithm. Although the set of abstract processors needn't exactly match the set of physical processors, there is a tacit assumption that abstract processors will be used at a similar level of "granularity" to physical processors. It wouldn't normally do to allocate abstract processors at the level of one processor per element of a program data array.

Remembering that a large overhead is usually incurred when a physical or abstract processor accesses data outside its own memory area, whereas local accesses are inexpensive, the logical rôles of the abstract processor arrangement are

- to define the level of data localisation—many important algorithms can exploit efficient local memory access at the abstract processor level, and
- to delimit evenly divided sets of subtasks, which can be executed in parallel.

On the other hand, there is a niche for a finer grain logical grid, which *is* targetted at the "array element" (or "problem grid") level. In HPF this rôle is assumed by the template.

For the remainder of this section we rely on the readers patience, to accept without further justification the utility of the template as a representation of a "problem-oriented grid". (This grid might, for example, correspond to some lattice of points over which a differential equation has been discretised.) In the next section we will see how data arrays are aligned to a template. Until then, the template will be treated as a sort of abstract array, with no associated data. Figure 9 represents the HPF data mapping scheme: the rest of this section is concerned with the bottom half of the diagram.

A template can be declared in much the same way as a processor arrangement.

```
TEMPLATE T (50, 50, 50)
```

delares a 50 by 50 by 50 three-dimensional template called `T`. Having declared it, we usually want to establish a relation between a template and some processor arrangement. We want to say in more or less detail how the elements of the template are distributed amongst the elements of the processor arrangement. This is done by giving `T` the `DISTRIBUTE` attribute[8].

As a first example, suppose we have

[8] This is Fortran-speak. In Fortran any variable has a type, which is best defined through an explicit type declaration. It may have other "attributes", such as shape. For historical

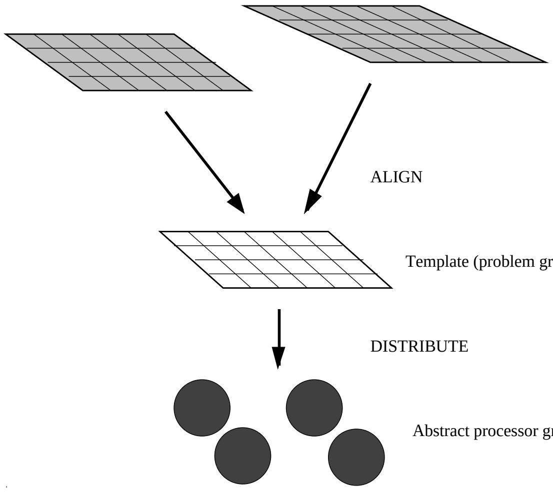

Figure 9: The stages of data mapping in HPF.

```
PROCESSORS P1 (4)
TEMPLATE T1 (17)
```

There are various ways in which `T1` may be distributed over `P1`. The four basic distribution formats are illustrated in figure 10. In the figure, each template element inhabits a particular memory area. This should be taken to mean that any data element aligned with a particular template element will be stored in the associated memory area (the template itself doesn't occupy any memory, of course).

Simple block distribution is specified by

```
DISTRIBUTE T1(BLOCK) ONTO P1
```

In this case, each processor gets a contiguous block of template elements. All processor get the same sized block, unless the number of processors doesn't divide the number of template elements, in which case the template elements are divided as evenly as possible (with the last processor holding fewer than the others, as illustrated).

Simple cyclic distribution is specified by

```
DISTRIBUTE T1(CYCLIC) ONTO P1
```

reasons attributes of a variable can be defined in separate statements to the type definition—the shape attribute can be defined in a separate `DIMENSION` statement. Fortran 90 adds a new clutch of allowed attributes. HPF extends the set again. In particular "distribution" is an attribute, in this sense, of a template.

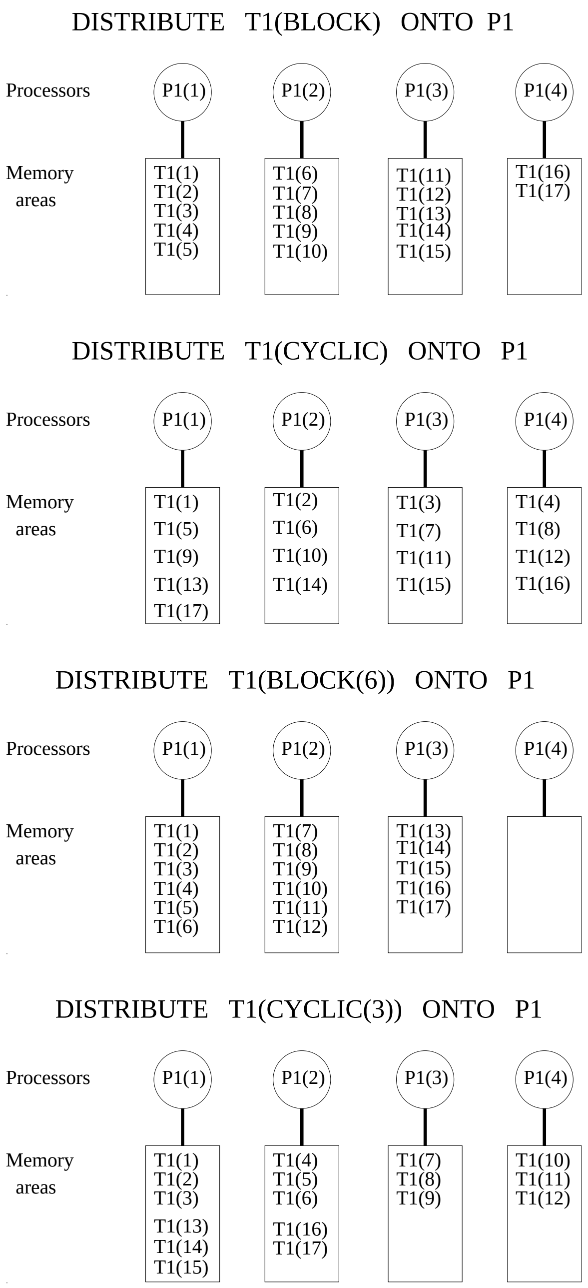

Figure 10: The four basic formats of template distribution

The first processor gets the first template element, the second gets the second, and so on. When the set of processors is exhausted, go back to the first processor, and continue allocating the template elements from there.

In a variant of the block distribution, the number of template elements allocated to each processor can be explicitly specified, as in

```
DISTRIBUTE T1 (BLOCK (6)) ONTO P1
```

If this means that we allocate all template elements before exhausting processors, some processors are left empty. It is *illegal* with to choose a block size here which cause template elements to be left over after all processors have had their blocks allocated. But in an analogous variant of the cyclic distribution ("block-cyclic distribution")

```
DISTRIBUTE T1 (BLOCK (3)) ONTO P1
```

the product of the number of processors with the block size can be smaller than the template size, and allocation wraps round after the first assignment of blocks to all processors.

That covers the case where both template and processor are one dimensional. When the template both have (the same) higher dimension, each dimension can be distributed independently, mixing any of the four distribution formats. The correspondence between the template and the processor dimension is the obvious one. In

```
PROCESSORS P2 (4, 3)
TEMPLATE T2 (17, 20)
DISTRIBUTE T2 (CYCLIC, BLOCK) ONTO P2
```

the first dimension of `T2` is distributed cyclically over the first dimension of `P2`; the second dimension is distributed blockwise over the second dimension of `P2`.

Finally, some dimensions of a template may have "collapsed distributions", allowing a template to be distributed onto a processor arrangement with fewer dimensions than the template. So

```
DISTRIBUTE T2 (BLOCK, *) ONTO P1
```

means that the first dimension of `T2` will be distributed over `P1` as was `T1` in the first example above. But for a fixed value of the first index of `T2`, all values of the second subscript are mapped to the same processor.

5.4 Data alignment

Some apologies are in order, for asking the reader to accept the idea of template distribution before giving any concrete examples of template usage. This we proceed to do.

Data is aligned to templates through the ALIGN directive. Before describing this directive, we return to one of the parallel examples given in section 3, for motivation. The core of the LU decomposition subroutine, around which the linear solver example was written, looks like this:

```
REAL A (N, N)
INTEGER N, R, R1
REAL, DIMENSION (SIZE (A, 1)) :: L_COL, U_ROW

DO R = 1, N - 1
  R1 = R + 1
  L_COL (R  : ) = A (R : , R)
  A (R , R1 : ) = A (R, R1 : ) / L_COL (R)
  U_ROW (R1 : ) = A (R, R1 : )
  FORALL (I = R1 : N, J = R1 : N)
    A (I, J) = A (I, J) - L_COL (I) * U_ROW (J)
  ENDFORALL
ENDDO
```

For simplicity, pivotting has been omitted here (pivotting improves the numerical stability of the algorithm, but the algorithm remains logically correct without it). The assignment to `U_ROW` has been rearranged slightly (for reasons which will be explained later). The final assignment to `A` has been changed to use a `FORALL` construct; the more opaque pure Fortran 90 version given earlier used simple array assignments and the intrinsic function `SPREAD`.

A cursory inspection of this algorithm should suggest that a possible choice of template for the problem is

```
TEMPLATE T (N, N)
```

This template exactly matches the main data structure of the problem, namely, the array `A` which holds the matrix. To align `A` to `T` we can add an `ALIGN` attribute declaration for `A` as follows

```
ALIGN A (I, J) WITH T (I, J)
```

In this context `I` and `J` act as *alignment dummies*. An alignment dummy is a named integer variable appearing as a subscript of the "alignee" (the array which is to be aligned) in an `ALIGN` directives. The idea is that the variable ranges over all possible index values associated with the array dimension involved. The "align target" (in this case the template `T`) can have expressions involving the alignment dummies amongst its subscripts. In this way, every element of the alignee is mapped to some element of the template.

In the example given above the mapping is very straightforward: it is the "identity mapping" between array and template. Much more general alignment relations are possible. We will illustrate a few, before returning to the main example above.

For one thing, indices can be permuted:

```
DIMENSION B (N, N)
ALIGN B (I, J) WITH T (J, I)
```

maps B to T, but in a "transposed" manner: B (1, 2) is mapped to T (2, 1), etc. More generally, a subscript of an *align target* (not an alignee) can be a linear expression in *one* of the alignment dummies (with integer expression coefficients). For example

```
DIMENSION C (N / 2, N / 2)
ALIGN C (I, J) WITH T (N / 2 + I, 2 * J)
```

Figure 11 illustrates how C would be aligned to T.

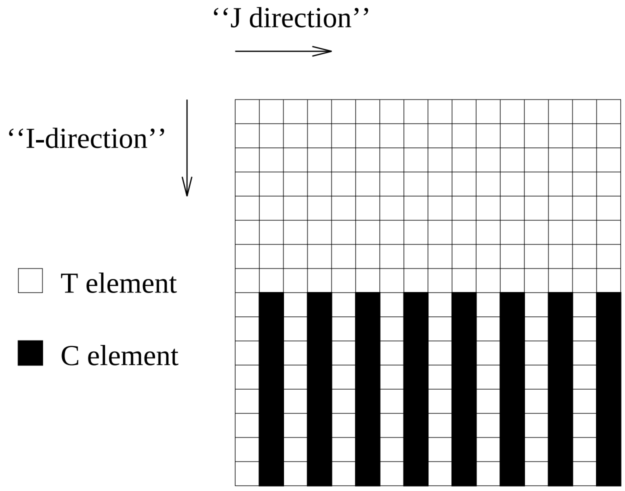

Figure 11: Example alignment of the array **C** to the template **T**

The rank of alignee and align-target do not have to match. A dimension of the alignee may be "collapsed"; or some dimension of the align-target might be a constant—for example a scalar might be aligned to the first element of a one-dimensional template; or the alignee may be "replicated" over some dimensions of the template. As an example of collapsing:

```
DIMENSION D (N, N, N)
ALIGN D (I, J, K) WITH T (I, J)
```

The template element is independent of K. This means that for fixed I, J, all elements of C differing only in their third index mapped to the same template element. Replication of array elements will be illustrated in the LU example below.

Before returning to that example, some general remarks on alignment. Firstly, restrictions on alignment: as mentioned, the expressions used to subscript a template must be linear in the alignment dummies, with integer coefficients; each subscript can depend on at most *one* of the alignment dummies (local to the ALIGN directive); and each alignment dummy can appear in at most *one* of the subscript expressions for the template. Secondly, we should say that there are various shorthand forms of the alignment directive which were not described here—they don't add any new facilities.

Returning to the main example, how should the other arrays in the program, L_COL and U_ROW be mapped? Most of the work in a single iteration is in the statement

```
      A (I, J) = A (I, J) - L_COL (I) * U_ROW (J)
```

We can minimise (indeed, eliminate) the need for communication in this assignment if L_COL (I) is stored wherever A (I, J), for any J is stored, and if U_ROW (J) is stored wherever A (I, J) for any J is stored. This can be specified by using a replicated alignment to the template T:

```
  ALIGN L_COL (I) WITH T (I, *)
  ALIGN U_ROW (J) WITH T (*, J)
```

where an asterisk as a template subscript means that data is replicated in the corresponding processor dimension[9]. Figure 12 is an attempt to visualise the alignment of the three arrays and the template.

As claimed, the main assignment in the FORALL construct will require no communications, because all operands of each elemental assignment will be stored on the same processor. What about the other statements?

```
  A (R , R1 : ) = A (R, R1 : ) / L_COL (R)
```

requires no communication. It is equivalent to

```
  FORALL (J = R1 : N)
    A (R, J) = A (R, J) / L_COL (R)
  ENDFORALL
```

and we know that L_COL (R) will be stored on any processor where A (R, J) is stored.

The other two array assignments *do* require communication. For example, the assignment to L_COL is equivalent to

```
  FORALL (I = R : N)
    L_COL (I) = A (I, R)
  ENDFORALL
```

[9] We specified *processor* dimension here, because it would be pointless to keep many copies of the data on a *single* processor, just because of a replicated alignment to a template dimension which had several index values mapped to the processor.

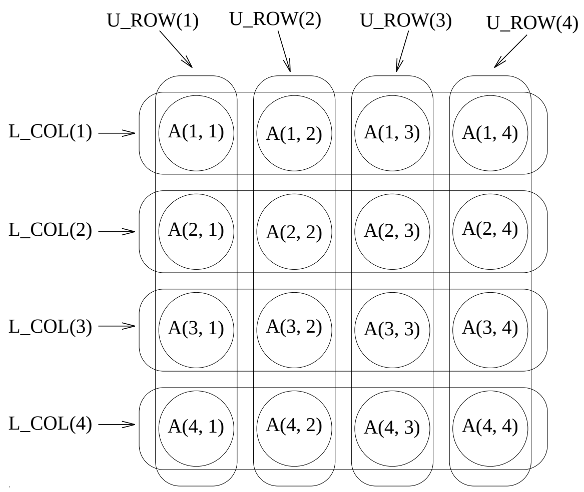

Figure 12: Alignment of the three arrays in the LU decomposition example

Because `L_COL (I)` is replicated in the "J" direction (as it has been consistently named above), whereas `A (I, R)` is held only on the processor holding the $J = R$ template element, updating the `L_COL` element involves broadcasting the `A` element to all interested parties. These communications the compiler will duly insert. Similarly, the `U_ROW` assignment involves broadcasts, this time in the "I" direction.

In the original Fortran 90 program the assignments to `U_ROW` and `A (R, :)` were done by

```
U_ROW (R1 : ) = A (R, R1 : ) / L_COL (R)
A (R , R1 : ) = U_ROW (R1 : )
```

The first assignment is equivalent to

```
FORALL (J = R1 : N)
  U_ROW (J) = A (R, J) / L_COL (R)
ENDFORALL
```

The problem here is that there are two strategies available to the compiler. It could allocate the computation on the RHS to the processor holding template element `T (R, J)` then broadcast the result in the I direction to assign to `U_ROW (J)`. This would reproduce the communication and computation pattern described above. Unfortunately, the compiler might alternatively broadcast `A (R, J)` and `L_COL (R)` in the I direction, *then* repeat the computation on every processor holding a copy of `U_ROW`. This is clearly a bad strategy, because it doubles the amount of communication, and unnecessarily introduces repeated computation. On would hope that a clever compiler wouldn't make this choice.

But compilers aren't always clever[10]. In situations like this it is probably wise to arrange assignments to *force* the compiler to do computations in sensible places, as the HPF version does.

Finally, we should comment on the distribution of the scalar "control" variables N, R, R1. These really want to be replicated across all processors. Because they are typically assigned in terms of globally replicated values, updating requires no communication. This is a case where we *can* tolerate replicated computation, because the assignments only happen occasionally. (One can see that it might difficult for a compiler to automatically recognise this fact, though). Although the HPF standard doesn't explicitly make the guarantee, a reasonable expectation is that (scalar) variables which don't have explicit mapping directives will be replicated globally. So we won't give an explicit mapping for these variables.

Having chosen a plausible alignment of our arrays to a template, we have to distribute that template. BLOCK distribution isn't particularly attractive for this computation, because successive iterations work on a shrinking area of the template. A block distribution could leave whole processors idle in later iterations. A CYCLIC distribution will achieve better load balancing (figure 13). Our final HPF program, distributed onto an M by M processor arrangement, is

```
INTEGER M, N, R, R1

PROCESSORS P (M, M)

TEMPLATE T (N, N)
DISTRIBUTE T (CYCLIC, CYCLIC) ONTO P

REAL A (N, N)
ALIGN A (I, J) WITH T (I, J)

REAL, DIMENSION (SIZE (A, 1)) :: L_COL, U_ROW
ALIGN L_COL (I) WITH T (I, *)
ALIGN U_ROW (J) WITH T (*, J)

DO R = 1, N - 1
  R1 = R + 1
  L_COL (R  : ) = A (R : , R)
  A (R , R1 : ) = A (R, R1 : ) / L_COL (R)
  U_ROW (R1 : ) = A (R, R1 : )
  FORALL (I = R1 : N, J = R1 : N)
    A (I, J) = A (I, J) - L_COL (I) * U_ROW (J)
  ENDFORALL
```

[10] The second strategy would be the one adopted by a system blindly following the celebrated "owner-computes" heuristic.

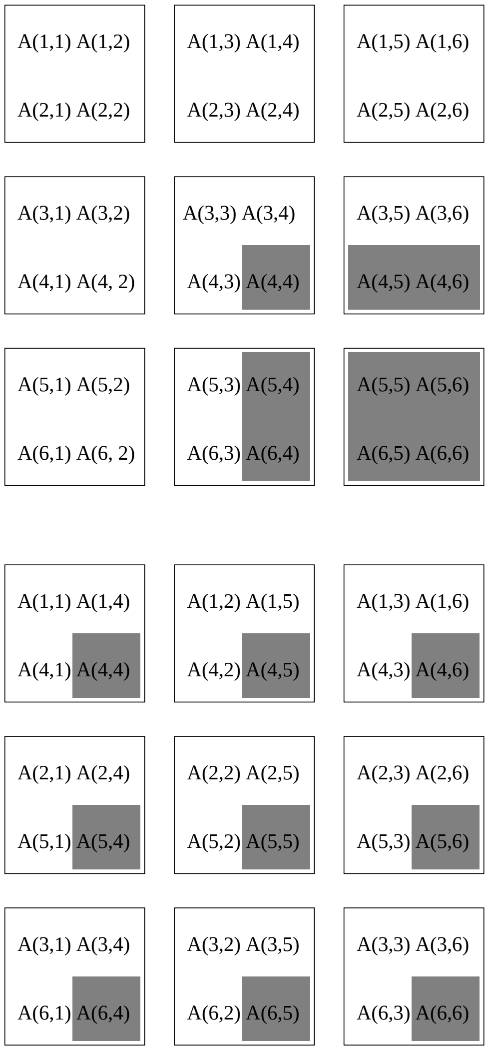

Figure 13: **A** elements involved (shaded) in iteration **R**=4, when **N**=6 and processor arrangement is 3 by 3. Block and cyclic distributions are illustrated.

```
ENDDO
```

5.5 Indirect alignments and implicit templates

The alignments described so far were all direct alignments to templates. It is also allowed to align an array to another array, using all the normal syntax of the `ALIGN` directive. So the alignments of `L_COL` and `U_ROW` in the example of the previous section could be written as

```
ALIGN L_COL (I) WITH A (I, *)
ALIGN U_ROW (J) WITH A (*, J)
```

In general, the meaning is that the alignee element is stored wherever the align-target array element is. Any such alignment *can* be written as an alignment to a template (provided the template is in scope). If there are several levels of indirection (array aligned to array aligned to array, etc) writing the "ultimate alignment" directly could be relatively complicated.

Also, templates can be left completely implicit, with DISTRIBUTE directives can applied directly to arrays. So, in the example, the `T` directives could be omitted, and we could add the directive

```
DISTRIBUTE A (CYCLIC, CYCLIC) ONTO P
```

The implicit template has the same shape as the array distributed. It is *not* allowed for any array to be both distributed in this way *and* to appear as an alignee in an `ALIGN` directive. So this idiom can always be regarded as shorthand for introduction of an explicit template.

These conventions are particularly useful for procedure dummy arguments, because templates can't be explicitly passed through the subprogram interface, as we will see in the next section.

6 The HPF subprogram interface

In a language like HPF, which handles explicitly distributed data, the subprogram interface is more or less bound to be troublesome. The underlying reason for this is clear—each distributed array is bundled with a complex set of mapping parameters, and structures such as templates. Getting all this information across the procedure interface in a flexible but efficient way is by no means straightforward.

The present definition of the HPF procedure interface leaves something to be desired. In some respects it seems over-complex, in others, less flexible than one might hope. In this document we try to concentrate on a useful subset of the defined interface, omitting some of its more awkward features.

6.1 Prescriptive mapping of dummy arguments

Conceptually the simplest approach to mapping of data in subprograms is for it to be completely decoupled from mapping in the calling program. The subprogram contains mapping directives for its local data and dummy arguments, following the syntax introduced in the last section. If a dummy argument has one mapping, and the corresponding actual has another, the compiler arranges (by means at it's own discretion) to remap the incoming data to agree with subprogram's expectation (and remap any outgoing results back to the callers expectation on exit from the subprogram).

This is called *prescriptive* mapping of the dummy arguments, because the mapping directives "prescribe" that data will be remapped.

```
PROGRAM MAIN
  PROCESSORS Q (2, 2)

  TEMPLATE U (4, 4)
  DISTRIBUTE U (CYCLIC, CYCLIC) ONTO Q

  COMPLEX M (4, 4)
  ALIGN M (I, J) WITH U (I)

  CALL HERM (M)
ENDPROGRAM

SUBROUTINE HERM (A)
  PROCESSORS P (4, 1)

  COMPLEX A (4, 4)
  DISTRIBUTE A (BLOCK, BLOCK) ONTO P

  FORALL (I = 1 : 4, J = 1 : 4)
    A (I, J) = CONJG (A (J, I))
  ENDFORALL
ENDSUBROUTINE
```

The actual and dummy argument of `HERM` have the same shape and type, so the call is certainly valid Fortran. They have completely different mappings (different processor arrangements, and `CYCLIC` versus `BLOCK`). The compiler transparently sorts this out, preserving the normal Fortran meaning of the program. (Incidentally, the mapping in the subprogram is an illustration of the use of an implicit template.)

The disadvantage to prescriptive mapping of dummies is the obvious one: there is a large run-time overhead in remapping the arguments. Although this penalty may be acceptable for large subroutines, such as those in a separatelycompiled numerical library, it often will not be acceptable for lightweight procedures in the users program.

6.2 Identification of processor arrangements, templates, and mappings

As a preliminary to discussion of parameter-passing methods which leave mapping of the arguments unchanged, we need to consider what it means for two mappings in different program units to be "the same". Fortran doesn't allow processor arrangements or templates to be passed as procedure arguments or included in common blocks, so there is a logical problem here—since mappings are defined by reference to processors and templates. In an HPF implementation based on full Fortran 90, this problem can be solved by putting processor arrangement or template declarations in a module, then importing this module into calling and called program units. Unfortunately, many early implementations of HPF probably won't incorporate the whole of Fortran 90, and modules may well be amongst the omitted features.

In practise, HPF adopts a model of "structural equivalence" between these objects. Two processor arrangements represent the same set of processors if and only if they have the same shape. We will say that two such processor arrangements are *identifiable* (with the caveat that this is not standard HPF terminology).

Likewise, we will say that two templates are identifiable if they have the same shape (they are conforming), and they have distribution directives with identical distribution formats, targetting identifiable processor arrangements.

Finally we have to say what it means for the mapping of two arrays to be "identifiable". This is more complicated[11]. The mappings of two arrays are identifiable if they have the same shape, and they have identical alignments to identifiable templates. This is fairly informal—to be more formal one would have to give precise rules for resolving indirection in alignment (when arrays are aligned to arrays), and for saying when ultimate alignments are identical. This is reasonably straightforward, and would make boring reading.

The rule in the last section is applicable to named array variables, with explicit alignment directives. It is also possible to define the alignment of, and thus identify mappings between, "regular sections" (array sections built without using vector subscripts). This is a useful facility, whose fairly obvious details we omit here.

[11] We will give a slightly stricter definition of identifiable mappings than the one implicit in the HPF definition. Some mappings with non-identifiable templates are regarded as being "the same" in the HPF definition. As admitted above, this discussion only covers a subset of HPF.

6.3 Descriptive and transcriptive mappings

A "descriptive" mapping of a dummy argument is one which *asserts* that an argument has a particular mapping, without implying any action to remap the argument on call of the procedure. In one respect, this is a very natural policy for argument mapping—it directly parallels the treatment of other argument attributes. In Fortran we don't expect that if the type of an actual is different from the type of a dummy, the types will be coerced to agree on entry and coerced back on exit. Instead we accept that if actual and dummy types disagree, the program is probably erroneous.

A descriptive mapping is flagged by the appearance of asterisks at appropriate points in the directives involved. A variant of the example in section 6.1, with descriptive alignment to an explict template (`MAIN` is unchanged from the previous version):

```
SUBROUTINE HERM (A)
  PROCESSORS P (2, 2)

  TEMPLATE T (4, 4)
  DISTRIBUTE T (CYCLIC, CYCLIC) ONTO P

  COMPLEX A (4, 4)
  ALIGN A (I, J) WITH *U (I)

  FORALL (I = 1 : 4, J = 1 : 4)
    A (I, J) = CONJG (A (J, I))
  ENDFORALL
ENDSUBROUTINE
```

The processor arrangements `P` and `Q` are identifiable; hence the templates `T` and `U` are identifiable; hence the mapping of the actual argument `M` is identifiable with the mapping of the dummy `A`. The asterisk preceding the alignment target means that this directive is descriptive. If the mapping of the dummy *hadn't* agreed with the the mapping of the actual, the program would have been erroneous.

Another variant, with descriptive distribution of an implicit template

```
SUBROUTINE HERM (A)
  PROCESSORS P (2, 2)

  COMPLEX A (4, 4)
  DISTRIBUTE A *(CYCLIC, CYCLIC) ONTO *P

  FORALL (I = 1 : 4, J = 1 : 4)
    A (I, J) = CONJG (A (J, I))
  ENDFORALL
ENDSUBROUTINE
```

There are two asterisks here, one for the distribution format clause, and one for the target processor arrangement clause (the reason for the separate treatment will become clear below).

The implicit template has the same shape as, a target processor arrangement identifiable with, and a distribution format identical to, the template of the actual argument. So the two templates are identifiable.

A variant of descriptive mappings allows clauses in the mapping directives to be "wild-carded". In this case, there is an asterisk but no explicit distribution format or target processor arrangement. The distribution directives in the previous example are could be changed to one of

```
DISTRIBUTE A * ONTO *P

DISTRIBUTE A *(CYCLIC, CYCLIC) ONTO *

DISTRIBUTE A * ONTO *
```

The first case asserts that the target of the distribution is `P` but says nothing about the distribution format. The second says that the distribution format is cyclic, but leaves the processor arrangement unspecified. The last provides no information about either[12].

When a directive contains these "wild-card" clauses (which are called "transcriptive" in the standard), the idea is that the mapping of the actual will be unchanged on entry to the subprogram, and the subprogram must be compiled to accept *any* legal variant of the wild-carded attribute.

There is no simple analogue of these transcriptive directives in the explicit template case. Instead there is a special attribute called `INHERIT`, which provides some of the functionality one might expect there. Its exact meaning is subject to interpretation, and this is not a suitable forum for speculation.

[12] Incidentally, it is also allowable for a `DISTRIBUTE` directive to have, say, a descriptive distribution format clause but a prescriptive target processor arrangement clause, or vice versa. The value of this facility is slightly unclear.