High Performance Fortran

Michael Metcalf
CERN, Geneva, Switzerland.

Abstract

This paper presents the main features of High Performance Fortran (HPF), a means to program data-parallel programs in a machine-independent way.

Keywords: Fortran, Parallel processing

1 HPFF

A basic problem in programming for parallel architectures is that each machine has its own design, and also its own specific software for accessing its hardware features. In 1992, as a response to this situation, the High Performance Fortran Forum (HPFF) was founded, under the guidance of Professor Ken Kennedy, to produce a portable Fortran-based interface to parallel machines for the solution of data-parallel problems, mainly involving regular grids. Over 40 organizations participated, and the work used existing dialects, such as Fortran D (Rice University), Vienna Fortran and CM Fortran (Thinking Machines), as inspiration.

It was realised early on that much of the desired functionality already existed in the, then, very new Fortran 90 (see [1]), and this was selected as the base language on which to build HPF itself. The array processing features of Fortran 90 are particularly relevant. This enabled the goal of producing an ad hoc standard for HPF within a year to be met, and full details can be found in [2]. The standard document itself is obtainable by anonymous ftp at titan.cs.rice.edu in the directory public/HPFF/draft as the file hpf-v10-final.ps.Z. At the time of writing, HPFF has reconvened to produce a second, more advanced, version – HPF II. One new feature is expected to be an extension for irregular grids.

The basic approach adopted was that of designing a set of directives that may be added to Fortran 90 programs, together with a few syntactical additions and some extra libraries, thus creating a data parallel programming language that is independent of the details of the architecture of the parallel computer it is run on. The principle is to arrange for *locality of reference* of the data by aligning related data sets to one another, and distributing the aligned sets over memory regions such that, usually, calculations on a given processor are performed between operands already on that processor. Any message passing that might nevertheless be necessary to communicate data between processors is handled by the compiler and run-time system.

2 Directives

The directives all have the form

```
   !HPF$ directive
```

and are interpreted as comment lines by non-HPF processors.

2.1 Alignment

There are various, sometimes quite complicated, ways of aligning data sets. A simple case is when we want to align three conformable arrays with a fourth:

```
   !HPF$ ALIGN WITH b :: a1, a2, a3
```

thus ensuring their subsequent distribution will be identical.

Although the ranks of the alignees must be the same, it is possible, using the '*' notation, to collapse a dimension so enabling the extents to differ:

```
   REAL a(3, n), b(4, n), c(43, n), q(n)
   !HPF$ ALIGN (*, :), WITH q :: a, b, c
```

where the ':' is a position holder for that dimension (taking elements in order). For the first dimension, the '*' causes the 3, 4 or 43 elements, respectively, to be aligned with q.

For single alignees, a statement form exists. This permits, additionally, a transpose via dummy variables (here j and k):

```
   !HPF$ ALIGN x(j, k) WITH d2(k, j)
```

as well as, in the following example, a lower bound to be fixed (first dimension of d), a dimension to be shifted (third dimension of d), or a stride to be defined (fourth dimension of d):

```
   !HPF$ ALIGN a(:, *, :, :, *) WITH d(31:, :, k+3, 2:8:2)
```

2.2 Distribution

Having aligned the data sets with one another, the next step is to map these data objects onto a set of abstract processors. Given

```
   REAL salami(10000)
   !HPF$ DISTRIBUTE salami(BLOCK)
```

we would, on a set of 50 abstract processors, map 200 contiguous elements to each one. This can be made more specific:

```
   !HPF$ DISTRIBUTE salami(BLOCK(256))
```

specifies the exact number per processor. The CYCLIC keyword is also available, to cycle the elements over the processors in turn.

For a multi-dimensional array, the methods may be combined:

```
   !HPF$ DISTRIBUTE three(BLOCK(64), CYCLE(128), *)
```

where, as before, the '*' collapses a complete dimension.

2.3 Processor layout

The layout of the abstract processors may be specified as a regular grid:

```
   !HPF$ PROCESSOR rubik(3, 3, 3)
```

and then distributions mapped onto it:

```
   !HPF$ DISTRIBUTE ONTO rubik :: a, b, c
```

Using a notation we have already seen, this may be further specified, as in this statement form

```
   !HPF$ DISTRIBUTE a(BLOCK, CYCLIC, BLOCK(3:19:4), *)     &
   !HPF$       ONTO rubik                              ! a is rank-4
```

For a high level of portability and efficiency, it is clearly necessary to be able to enquire about the actual processor layout. For this, two new intrinsic functions provide the number of processors and the actual shape of their layout. Thus, the abstract layout may be specified in terms of the actual number available:

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

and an array, here ps, may be defined to hold the shape of the layout, each element of ps containing the number of processors in the corresponding dimension of the layout:

```
   INTEGER, DIMENSION(SIZE(PROCESSORS_SHAPE())) :: ps
   ps = PROCESSORS_SHAPE()
```

2.4 Templates

Usually, we align arrays to one another in such a fashion that at least one of them covers the entire index space of all of them, as in

```
!HPF$ ALIGN a WITH b
```

Where it is required to make arrays partially overlap in some fashion, it would be possible to use an artificial array to support the mapping. However, after much debate, HPFF decided to incorporate this facility into the HPF language using the TEMPLATE directive. Its use is shown in

```
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK, BLOCK) :: earth(n+1, n+1)
REAL, DIMENSION(n, n) :: nw, ne, sw, se
!HPF$ ALIGN nw(i, j) WITH earth(i  , j  )
!HPF$ ALIGN ne(i, j) WITH earth(i  , j+1)
!HPF$ ALIGN sw(i, j) WITH earth(i+1, j  )
!HPF$ ALIGN se(i, j) WITH earth(i+1, j+1)
```

where each of the four alignees, nw, ne, sw and se, is mapped to a different corner of the template, earth.

2.5 Dynamic alignment and distribution

The directives described so far have all had effect at compile time. By contrast, the DYNAMIC attribute:

```
!HPF$ DYNAMIC a, b, c, d
```

or

```
!HPF$ DYNAMIC, ALIGN WITH s :: x, y, z
```

allows the use, at run time, of the REALIGN and REDISTRIBUTE statements. These are similar to the corresponding directive forms, but the rules of Fortran 90 allow a more general form of subscript expressions.

3 Parallel constructs

The parallel constructs are mostly extensions to the Fortran 90 syntax. The Fortran standardization committees are likely to add these to Fortran 95, a minor revision of Fortran 90 now in preparation.

3.1 FORALL statement and construct

The FORALL is an addition to the Fortran 90 syntax that assures a compiler that the individual assignments in a statement are independent, and can therefore proceed in parallel. It also overcomes some restrictions found in ordinary array assignments, in particular that the left-hand and right-hand sides of assignments must be conformable arrays. Examples of the statement form are:

```
FORALL(i = 1:n, j = 1:m)                  a(i, j) = i + j
FORALL(i = 1:n)                           a(i, i) = x(i)
FORALL(i = 1:n, j = 1:n, y(i, j) /= 0.) x(j, i) = 1.0/y(i, j)
```

The construct form allows, in addition, a sequence of independent statements to be executed in order and once only. In

```
FORALL(i = 2:n-1, j = 2:n-1)
   a(i, j) = a(i, j-1) + a(i, j+1) + a(i-1, j) + a(i+1, j)
```

```
      b(i, j) = a(i, j)
   END FORALL
```
the second assignment will not begin until the first has completed for all values of i and j, and will then use the newly computed values.

3.2 PURE attribute

An obstacle to generating parallel code in the presence of function references is that non-intrinsic functions may have side effects that potentially change the results of subsequent assignments. Within a FORALL statement or construct, the programmer is able to make a pact with the compiler, asserting that the function referenced has no side effects and may be safely referenced in parallel invocations. This is achieved by giving such functions the PURE attribute, a further Fortran 90 syntax extension. Given
```
   PURE FUNCTION my_func(j)
```
we can invoke
```
   FORALL(i = 1:n) a(i) = my_func(i)
```
We are saying that my_func does nothing other than return a result, and in particular that it does not change the value of its argument, performs no I/O, and modifies no global variable (e.g. in a module).

3.3 Parallel loops

Unless it can determine otherwise by dependency analysis, a compiler has to make the assumption that the individual statements of a DO or FORALL construct depend on one another. It is possible in HPF to insert a directive that asserts that each iteration or statement is, in fact, independent of all others, as in
```
   !HPF$ INDEPENDENT
      DO i = 1, 100
         a(p(i)) = b(i)           !p is a permutation
      END DO
```
where, as p is a permutation, all assignments are independent and can proceed in parallel. In nested loops, each one requires its own directive, where appropriate.

4 HPF intrinsic and library procedure

The Fortran 90 intrinsic functions are augmented by a further three for use in a parallel environment, and by an HPF Library of procedures. Their large number means they cannot be described here, and the interested reader is referred to the standard or to [2]. Suffice it to list their principal groupings: to determine array mappings, additional bit manipulation functions, additional array reduction functions, array sorting, array scatter functions, and two sets of partial array reduction functions.

5 Extrinsic procedures

HPF introduces the notion of extrinsic procedures. This defines both an interface to non-HPF procedures, or even languages, and a mechanism for implementing the SPMD programming model. For this latter purpose it is possible to pass parts of a decomposed array to local procedures on each processor, and the extrinsic procedure thus defined terminates when each local copy has finished executing on its own part of the array. Any communications require explicit management.

6 Storage association

FORTRAN 77 is based on storage association through the use of, in particular, COMMON and EQUIVALENCE. Use of these facilities makes parallelization of programs difficult, as otherwise unrelated variables may appear to be related because they are, for instance, in the same COMMON block. Fortran 90 provides new facilities for avoiding storage association, and so it should not be used in new programs. However, for legacy programs, the NOSEQUENCE directive is provided, asserting that certain apparent dependencies do not, in fact, exist.

7 Subset HPF

In order to enable useful implementations of HPF to be produced quickly, a subset language was defined. This consists of Fortran 90 apart from modules, derived data types, the CASE construct, etc., plus the HPF extensions without dynamic realignment and distribution, certain complicated alignments, the PURE attribute, the FORALL construct, the HPF Library, and extrinsic procedures. Early users are advised, initially, to stick to this subset language.

8 Conclusion

This paper has briefly described the main features of HPF, in particular those for aligning and distributing regular grids of data over the processors of a parallel computer. HPF-conformant compilers are now available, for instance from DEC and IBM, and other vendors offer HPF preprocessors. However, as far as high-energy physics is concerned, we note that only within the area of accelerator design do there appear to be suitable applications for using HPF. Application programs for processing experimental data do not contain the large regular grid patterns that HPF is designed to handle. We can thus expect very little use of HPF in our field.

References

[1] M.Metcalf and J.Reid, *Fortran 90 Explained*, Oxford U. Press, Oxford (1990).

[2] C. Koebel et al., *The High Performance Fortran Handbook*, MIT Press, Cambridge, MA, (1994).