Discussing HPF Design Issues

(TR A-284, to appear in Europar'96)

Fabien COELHO (coelho@cri.ensmp.fr)

Centre de Recherche en Informatique, École des mines de Paris,
35, rue Saint-Honoré, F-77305 Fontainebleau CEDEX, FRANCE.
Phone: +33 1 64 69 48 52. URL: http://www.cri.ensmp.fr/pips

Abstract. As High Performance Fortran (HPF) is being both developed and redesigned by the HPF Forum, it is important to provide comprehensive criteria for analyzing HPF features. This paper presents such criteria related to three aspects: adequacy to applications, aesthetic and soundness in a language, and implementability. Some features already in HPF or being currently discussed are analyzed according to these criteria. They are shown as not balanced. Thus new or improved features are suggested to solve the outlined deficiencies: namely a scope provider, multiple mapping declarations and simpler remappings.

1 Introduction

Analyzing and discussing the design of a programming language [14] involves balancing issues related to targeted applications and possible implementations within a compiler. The design must also address the technical and aesthetic quality of features. HPF provides a data parallel programming model for distributed memory parallel machines. Users advise the compiler about data mapping and parallelism through directives, constructs and intrinsic functions added on top of Fortran 90 [9]. The compiler is expected to switch from the global name space and implicit communication of HPF to the distributed local name space and explicit communication of the target parallel machine. This paper studies HPF design issues, after three years of existence and as commercial compilers have been released. The language is being both developed and redesigned by the HPF Forum: new features are considered for addition and some are selected for defining an efficiently implementable kernel. There is indeed an opportunity to improve HPF on the application, language and implementation aspects, as features are questioned at light of experience.

HPF version 1.1 [6] includes directives for describing regular mappings onto multiprocessor machines, parallel constructs, and intrinsics considered as useful or expressing some parallelism. Array mappings are statically declared through `template` and `processors` objects using `align` and `distribute`. Arrays with the `dynamic` attribute may be changed their mapping at runtime with `realign` and `redistribute` *executable* directives. Subroutine arguments may be specified a *descriptive*, *prescriptive* or *transcriptive* mapping: it may be warranted, to be enforced or unknown with `inherit`. Parallel loops are tagged with `independent`. `new` specifies variables private to iterations. `forall` is a data-parallel construct which generalizes Fortran 90 array expressions. Intrinsics are available to query at runtime about data mappings and to specify reductions.

These features form the core of HPF. After three years of existence, they are reconsidered at the light of early compiler implementations, application ports or teaching sessions. The Forum is redesigning the language to allow efficient implementations. This paper presents criteria to analyze HPF features (Section 2). Some current and planed features are then analyzed with respect to these criteria (Section 3). They are shown as not balanced. This is the ground for new or improved HPF features which are suggested (Section 4) before concluding.

2 Analysis criteria

It seems natural to analyze language features with respect to the application, language and implementation aspects which are respectively the concern of users, designers and implementors. For each point, several issues are discussed, which express the interests each group might find in a feature.

2.1 Application

The preliminary goal of HPF is to suit data-parallel applications to be run efficiently on distributed memory parallel machines. HPF feature benefits for users can be seen through three perspectives.

Useful: some applications require or benefit from a feature for efficiency. Some unconstrained features such as remappings are not useful in their full generality and are difficult to handle for the compiler.

Expressive: HPF aims at providing to the user a mean to express its knowledge about the application in order to help the compiler. If some knowledge cannot be expressed while it would be useful, the language lacks something.

Elegant: notations must be concise, intuitive and as simple as possible. Thus they are easy to teach, learn and remember. Also users may misunderstood some features which are error-prone.

2.2 Language

From the designer point of view, language features must be neat, and have some technical properties which are considered as qualities, or their absence are considered as design mistakes and source of difficulties for users. It is

Homogeneous if different features share a common look and feel. This issue is illustrated in Fig. 1, where two homogeneous directive styles are presented. The first one applies on the next statement and is verbose, while the other is external and applies on the explicitly specified variables.

Orthogonal if features are not constrained one with respect to the other. HPF is built orthogonaly (with very few restrictions) on top of Fortran 90: it is up to the compiler to insure that HPF and Fortran features interact as expected. Also for instance Fortran 90 array expressions cannot specify a transposition, because dimensions are taken in the same order. HPF `forall` provides the array expression and dimension orthogonality.

Complete if all cases are expressed. The `inherit` mapping addresses this point. While this is a nice property for a language, handling any case with no motivation for efficiency can have a significant impact on the compiler costs, if not on the compiler structure.

```
!hpf$ independent
      do k=1, n
!hpf$   independent
        do j=1, n
!hpf$     new
          s = 0.
!hpf$     independent
          do i=1, n
!hpf$       new
            t = b(i,j,k)+c(i,j,k)
            a(i,j,k) = t*t+t
!hpf$       reduction
            s = s + t
!hpf$       reduction
            p = p * t
          enddo
!hpf$     reduction
          p = p * s
        enddo
      enddo
```

```
!hpf$ independent(k,j), &
!hpf$   new(s), reduction(p)
      do k=1, n
        do j=1, n
          s = 0.
!hpf$     independent(i), &
!hpf$       new(t), reduction(s)
          do i=1, n
            t = b(i,j,k)+c(i,j,k)
            a(i,j,k) = t*t+t
            s = s + t
            p = p * t
          enddo
          p = p * s
        enddo
      enddo
```

Fig. 1. Internal and external directive styles

2.3 Compiler

The last but not least issue is the implementability at a reasonable cost in a compiler. The state of the art evolves continuously because of research, but general conditions for sound and simple implementations remain. As the market for the language is not very large development costs due to useless features must be avoided. For the implementor a feature must be

Simple: features are added with *some* possible use in mind, but implementors must handle them in *all* cases, including the worst; it may also happen that some future optimizations are prevented or becomes more difficult because of complex features; this issues are illustrated below with remappings.

Efficient: to allow compilation to produce efficient code, as much static information as possible must be provided, and compile-time handling must not be prevented; see the `assume` directive suggestion in Section 4.1.

Separable: as interprocedural compiler technology [8] is not yet the common lot of software providers, features must not prevent separate compilation of subroutines; Fortran 90 explicit subroutine interfaces provide the mean to allow both maximum information about callees and separate compilation, if they are both mandatory and extended.

3 HPF feature analyses

Let us now analyze some HPF features according to these criteria. They are shown either as not simply and efficiently implementable or miss some instances. The difficulty of the design is to balance criteria for a given feature, in order to reach a sound proposal.

```
      subroutine callee(A)
      real A(n)
!hpf$ inherit A
      ! A mapping is not known!
       A = ...
```

Fig. 2. Inherit unknowledge

```
!hpf$ distribute A(block)
      if (...) then
!hpf$   redistribute A(cyclic)
      endif
      ! A is block OR cyclic
      A = ...
```

Fig. 3. Remapping unknowledge

Inherit was added for completion purposes when specifying mappings of subroutine arguments. Some passed arguments cannot be described naturally with descriptive mappings, thus the need to be able to specify something else. Compilation of this feature is tricky because the compiler cannot assume any knowledge about the mapping (Fig. 2). Applications in which different mapping may reach a given subroutine tend to use this non-description while some knowledge could be available. Section 4.1 discusses the `assume` directive which can provide more knowledge to the compiler.

Remappings are well integrated in HPF, but not in compilers: they are definitely useful to application kernels such as ADI [10], FFT [7] and linear algebra [12]; the syntax is homogeneous to static mapping directives; remappings are orthogonal to the program control flow. This very orthogonality is the nightmare of implementors (if any): remappings may appear anywhere in the program, and may depend on runtime values. Thus for a given array reference the compiler may not know the mapping status of an array, as show in Fig. 3. This ambiguity similar to `inherit` has a strong impact on the way references can be handled, delaying everything to runtime, and preventing standard analyses and optimizations from giving their full benefice. This worst case must be handled before optimizing for programs where there is no such ambiguity. Section 4.2 discusses possible improvements in the design.

Internal style reductions (Fig. 1) were first considered by designers [13], and then changed [3]: although equivalent to the external syntax, this choice was not homogeneous with the `new` directive, which also deals with scalars. Moreover the external syntax eases the implementation: in the example the reduction on `p` is over the full loop nest while the one on `s` is only at the `i` level. This level is directly specified with the external syntax, while it must be extracted upward in the code checking for some condition with the internal syntax, requiring more work from the compiler.

Independent, New and Reduction are definitely not well integrated in the language: the directive styles are not homogeneous (Fig. 1): it is internal for `independent` and external for others. Moreover these directives are not orthogonal: `new` and `reduction` must be attached to an `independent`. However some non independent loops could benefit from these directives, as well as pieces of flat code. Section 4.3 develops an homogeneous and orthogonal proposal. This proposal does not change implementability issues, but allows more information to be given to the compiler.

```
assume-directive =
   ASSUME
     spec-directive-or-list
   END [ASSUME]

spec-directive-or-list =
   spec-directive-list
 [ OR
   spec-directive-or-list ]

spec-directive-list =
   <nothing>
 | spec-directive
 [ spec-directive-list ]
```

Fig. 4. `ASSUME` directive proposal

```
attached-directive =
   explicit-independent-directive
 | new-directive
 | reduction-directive
 | ...

explicit-independent-directive =
   INDEPENDENT(...)

scope-directive =
   BLOCK
   statements-and-directive-list
   END [BLOCK]
```

Fig. 5. Scope for orthogonal directives

```
      subroutine callee(A(100))
!hpf$ ASSUME
!hpf$   distribute A(block)
!hpf$ OR
!hpf$   distribute A(cyclic)
!hpf$ OR
!hpf$   ! nothing, not mapped
!hpf$ END ASSUME
```

Fig. 6. Assume example

```
!hpf$ DISTRIBUTE A(block,block)
      if (...) then
!hpf$   REDISTRIBUTE A(*, block)
        ...
      endif
! A is (*,block) or (block,*)
! thus must *not* be referenced
!hpf$ REDISTRIBUTE A(*,block)
```

Fig. 7. Remapping example

4 Improved and additional features

In the previous analysis, several deficiencies of HPF features were outlined: lacks of simple implementability, language homogeneity and orthogonality or expressiveness for some applications. In the following, several new or improved features are suggested to solve these problems.

4.1 Assume

HPF does not allow to describe several mappings that may reach subroutine arguments. Thus either `inherit` is used, warranting inefficiency, and/or subroutines are cloned by hand to build several versions to be compiled with more knowledge. There is a tradeoff between efficiency and software quality. The `assume` directive [2, 5] outlined in Fig. 4 allows to describe several mapping for arguments in the declaration part of the subroutine. Fig. 6 shows a simple example. It asserts that array `A` may reach the subroutine with a block or cyclic distribution, or no distribution at all (*i.e.* a local array).

It is up to the compiler to manage the cloning and selection of the right version. Some carefully designed constraints are necessary for allowing both simple and efficient implementations: (1) the `assume` directive must also appear in the explicit subroutine interface; (2) the subroutine argument mappings must appear in the same order in both declarations; These two points all together allow separate compilation and compile/link-time handling of the feature.

The expressiveness is maximum, because no constraints are needed on the mapping directives involved: descriptive and prescriptive mappings can be specified for the arguments, as well as on local variables, including partially redundant or useless specifications. Thus no special checking is required. Several and nested `assume` directives allows to describe the cross product of possible mappings. Constraints (1) and (2) allow callers and callees to agree about naming conventions for efficient and separate compile and link time implementation: when compiling a call, the list of assumed mappings is scanned for the best match and this version is called. Its name should depend on the version number, which is the same in both caller and callee declarations. From these constraints, a program with an assume specification can be translated by cloning into a standard HPF program as described in [5].

4.2 Remappings

As shown above remappings are perfectly integrated in the language and suited to some applications. It is perfect, but the implementation. Thus the current design of the HPF kernel excludes all remappings. We argue [2] that some could be kept, and especially those that are the most useful. This suggestion is supported by a full implementation in our prototype HPF compiler [4].

The following constraints are suggested: (1) all remappings must be statically known (*i.e.* as constrained as static mapping declarations); (2) array references should have at most one reaching mapping. These constraints disallow all ambiguities at the price of orthogonality: from (1) the mapping cannot depend on runtime values, and (2) implies that remappings cannot appear anywhere. Checking for (2) is similar to the the reaching definition standard data flow problem [1]. This second restriction does not apply to remapping statements: there may be some points in the program where the mapping of an array is ambiguous, provided that the array is not referenced at this point. This is illustrated in Fig. 7: between the `endif` and up to the last `redistribute` the array mapping is ambiguous to the compiler, thus the array must not be referenced. In such cases the runtime must keep track of the current status of an array in order to perform the right remapping. Remappings are much easier to compile under these constraints.

These restrictions do not prevent any use of remappings in the real application kernels cited in Section 3. Indeed, it is very hard to imagine a real application in which the desired mapping of some references would be unknown to the user. As computations are performed by the application, the user has a clear idea of the right mapping at different point and the remapping directives naturally reflect this information to the compiler without ambiguity. The pathological cases that are the nightmare of implementors are not those that would appear in any real application. Thus the suggested constraints, while allowing a much simpler implementation, does not prevent the use of remappings for efficiency.

```
!hpf$ new(time), &
!hpf$   reduction(s)
      do i=1, n
        s = s + ...
        time = i*delta
        ! some use of time
      enddo
      ! no use (read) of time
```

Fig. 8. Orthogonality of directives

```
!hpf$ new(t), reduction(s)
!hpf$ block
      t = a(1,1)+a(2,1)
      a(1,2) = t*t
      s = s + t ! out of a loop
      do i=1, n
        s = s + ...  ! in
        ! some computation
      enddo
!hpf$ end block
```

Fig. 9. `new` and `reduction` scoping

4.3 Scope

As noted above, the `independent`, `new` and `reduction` directives are not homogeneous and orthogonal. Let us first allow for homogeneity the external syntax for `independent`. Fig. 8 shows a loop example where the non-orthogonality prevents the assertion of useful information: if iterations are performed across processors in the first example, `new` advises the compiler not to transmit and update the `time` variable between all processors, and `reduction` does the same for `s`, avoiding useless communication. These directives may also be useful outside of loops, as shown in Fig. 9 (with the syntax suggested hereafter): stencil computations typically use particular formula for corners that use temporary scalars which can be private to the processor holding the corner.

The scope proposal in Fig. 5 provides a solution to these problems. Directives are orthogonal one to the other, and apply on the next statement scope, which may be a loop or a scope-provider directive in the spirit of what is suggested for the `on` clause [11]. The `block/end block` provides scoping for flat regions of code. It would be obsoleted by the addition of such a structure to Fortran.

For consistency and homogeneity, these directives must be active for the next statement scope. Thus `independent(i)` implies that all loops with index `i` in the scope are parallel. Parallel loop indexes are necessarily and implicitly private. Moreover `new` and `reduction` do not make sense on `forall`, because no scalar can be defined within this construct. Scopes may be nested: in Fig. 1 scalar `s` is private to the full loop nest but is protected as a reduction in the inner loop. This proposal gives more homogeneity and orthogonality to HPF. Implementation issues are not changed: more information than before is available to the compiler, that can only benefit to applications.

5 Conclusion

HPF design has been discussed with respect to application, language and implementation issues. Several features have been analyzed and deficiencies outlined. Such an analysis is obviously much easier after experience. Then new features were suggested to improve the language with respect to our criteria. An `assume` directive, a new scope provider and simpler remappings were described. Possible simple and efficient implementation techniques were presented.

Other HPF features could also be discussed: replication of data through `align` seems tricky for users. The simpler and intuitive syntax suggested in Fortran D and Vienna Fortran could allow this specification in `distribute`. The `on` directive currently discussed is in the internal style spirit and includes a private scope provider. This does not improve HPF homogeneity and orthogonality.

Discussing design issues is a difficult task. Features are developed and discussed independently one of the other, thus homogeneity issues and possible interactions are easily overlooked in the difficult process of writing a sound and correct specification. Possible alternatives to design decisions are not obvious. Although a Forum may be able to select good ideas once expressed and formalized, such ideas are not always found. Some problems come only to light after experience. Teaching, implementing and using the language helps to clarify many points, to identify new problems and to offer new solutions.

Acknowledgments: Charles KOELBEL, Robert SCHREIBER, Henk SIPS and Henry ZONGARO for electronic discussions; Corinne ANCOURT, Denis BARTHOU, Béatrice CREUSILLET, Ronan KERYELL and the referees for comments.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.
2. F. Coelho. Discussion about HPF kernel/HPF2 and so. `hpff-distribute` list, December 13, 1995.
3. F. Coelho. HPF reduce directive. `hpff-task` list, January 2, 1996.
4. F. Coelho and C. Ancourt. Optimal Compilation of HPF Remappings. CRI TR A 277, CRI, École des mines de Paris, Oct. 1995. To appear in JPDC, 1996.
5. F. Coelho and H. Zongaro. `ASSUME` directive proposal. TR A 287, CRI, École des mines de Paris, Apr. 1996.
6. H. P. F. Forum. *High Performance Fortran Language Specification*. Rice University, Houston, Texas, Nov. 1994. *Version 1.1*.
7. S. Gupta, C.-H. Huang, and P. Sadayappan. Implementing Fast Fourier Transforms on Distributed-Memory Multiprocessors using Data Redistributions. *Parallel Processing Letters*, 4(4):477–488, Dec. 1994.
8. F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *ACM International Conference on Supercomputing*, pages 144–151, June 1991.
9. ISO/IEC. *International Standard ISO/IEC 1539:1991 (E)*, second 1991-07-01 edition, 1991.
10. K. Kennedy and U. Kremer. Automatic Data Layout for High Performance Fortran. CRPC-TR94 498-S, Center for Research on Parallel Computation, Rice University, Dec. 1994.
11. C. Koelbel and R. Schreiber. ON clause proposal. `hpff-task` list, January 8, 1996.
12. L. Prylli and B. Tourancheau. Efficient block cyclic data redistribution. Research Report 2766, INRIA, Jan. 1996. To appear in Europar'96.
13. R. Schreiber, C. Koelbel, and J. Saltz. reductions. `hpff-task` list, January 2, 1996.
14. B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley Publishing Company, 1994.