`ASSUME` Directive Proposal (TR A-287)

Fabien Coelho
École des mines de Paris
coelho@cri.ensmp.fr

Henry Zongaro
IBM Canada
zongaro@vnet.ibm.com

May 15, 1996

Abstract

This proposal allows the declaration of several mappings instead of just one at subroutine boundaries. Markers within subroutine declarations separate different argument mapping assumptions provided to the compiler. Full HPF specification directives can be expressed. A rewriting-based implementation technique is also suggested to translate such a program into a standard HPF program.

1 Motivation

Subroutine dummy arguments can be given *descriptive*, *prescriptive* or *transcriptive* (`inherit`) mappings through HPF specification directives, or no mapping at all. While descriptive and prescriptive are thought as efficiently manageable, transcriptive can lead to very poor performances because no static knowledge is provided to the compiler.

If several actual argument mappings can reach a subroutine, such information cannot be expressed to the compiler. An alternative to the poor performances of `inherit` in such cases is hand cloning: each routine is then compiled with some direct descriptive or prescriptive mappings, and the right version must be called, maybe through some runtime switching. This leads to poor software engineering practices.

This proposal aims at allowing the specification of several mappings that may reach a routine, with minimum addition to the HPF mapping semantics. Another issue is to preserve separate compilation. A very important point which drives this proposal is its implementability and efficiency with the current compiler technology, while preserving maximum flexibility and expressiveness for users.

2 Proposal

The idea is to specify a set of mappings instead of one mapping for subroutine dummy arguments. Each subset should be consistent on its own; that is the specification should make sense for the compiler. It can be seen as some kind of interface to the caller in a callee, as there are interfaces about callees in callers.

2.1 Full syntax

The syntax of the assume directive consists of markers to distinguish between different standard HPF specification directive sets to be taken into account by the compiler. Each line with the assume directive scope should appear after a *directive-origin* or be a comment. The syntax is the following:

assume-directive **is**
  `ASSUME`
  list-of-or-separated-specification-directives
  `END ASSUME`

list-of-or-separated-specification-directives **is**
  list-of-specification-directives
  [`OR`
  list-of-or-separated-specification-directives]

list-of-specification-directives **is**
  [*specification-directive*
  [*list-of-specification-directives*]]

Constraint 1 *The assume-directive may appear within the declarations of the subroutine. If so, it* must *also appear in the subroutine explicit interface, the mapping specifications related to dummy arguments being in the very same order. (The rationale for this is to allow both caller and callee to aggree about naming conventions if a simple rewriting-based technique is used for implementing the feature).*

Constraint 2 *A variable with the `SAVE` attribute or which is a member of a common must not be specified several mappings through the assume-directive (thus these variables may appear in an assume-directive, provided that they are mapped the very same way in each set of specifications, what means that their mapping specifications could be factorised out of the assume-directive).*

Each *list-of-specification-directives* in the *list-of-or-separated-specification-directives* specify a distinct set of assumptions about mappings that may reach the subroutine arguments. The compiler can take advantage of the provided information as described in Section 4.

2.2 Remarks

Note 1 *The list-of-specification-directives may be empty, specifying that the arguments are not explicitely mapped.*

Note 2 *The list-of-specification-directives may include directives on variables local to the subroutine.*

Note 3 *Arbitrary mappings (*descriptive, prescriptive *or even* transcriptive*) can be specified within the assume scope, including not only alignments, distributions, but also processors and template declarations.*

Note 4 *There may be other specification-directives outside the scope of the assume-directive.*

Note 5 *There may be several assume-directives within the declarations of the subroutine, specifying the cross product of the specifications. For instance both specifications in Figure 1 are equivalent.*

Note 6 *assume-directives may be nested, specifying the cross product of the specifications. For instance both specifications in Figure 2 are equivalent.*

3 Example

Let us consider the example in Figure 3. Program `assume` calls subroutine `iterate` in four different contexts. The assume directives allows the caller and callee to agree upon the possible mappings of arguments. Maximum static knowledge is provided to the compiler, for both handling the caller and the callee.

- program `assume`
  - four local arrays are declared, of which three are distributed.
  - subroutine `iterate` interface is declared, which uses a nested assume directive; it includes a `processors` declaration.

```
ASSUME
  d1
OR
  d2
END ASSUME
ASSUME
  d3
OR
  d4
END ASSUME
```

```
ASSUME
  d1
  d3
OR
  d1
  d4
OR
  d2
  d3
OR
  d2
  d4
END ASSUME
```

Figure 1: Several *assume-directive*s

```
ASSUME
  ASSUME
    d1
  OR
    d2
  END ASSUME
  d3
END ASSUME
```

```
ASSUME
  d1
  d3
OR
  d2
  d3
END ASSUME
```

Figure 2: Nested *assume-directive*s

  - three possible reaching mappings are specified, of which two are prescriptive and the last one is empty; thus it specifies three calling contexts: block, cyclic and not distributed.
  - The first three calls to `iterate` correspond to the three contexts: no remapping is required at runtime.
  - For the last call, the compiler must chose which *version* to call. Both prescriptive mappings are candidate. The compiler may chose the first one, remapping array `D` to block, or apply any criterion, for instance the expected cost due to the remapping.

- subroutine `iterate`
  - one local array (`TMP`) is declared.
  - a saved variables is also declared (`count`)
  - the assume directive specifies the mapping of the local array and dummy argument.
  - the same three reaching mappings are specified.
  - it would not make much sense to have the `TMP` to `X` alignment or the `processors` declaration outside of the assume because there are no distribution for the third case.

4 Advice to implementors

In this section we describe a possible implementation based on cloning. Such an implementation technique does not put new requirements on the compiler technology: the cloning procedure suggested hereafter is a rewriting technique applied on an assume specification to match a standard HPF program, thus can be applied with the current state of the art in compilation. The contraints specified on the assume specification allow this technique to be applied in a separate compilation context. This technique is only illustrative.

Compilation of the callee

The callee is compiled by cloning the subroutine for each specified mapping set. For the example in Figure 3 it would lead to three subroutines. Some naming convention must be chosen so as to differentiate the different versions. Let us assume it consists of appending $\_i$ to the subroutine name,

```
  program assume
    real, dimension (100) :: A, B, C, D
!hpf$ processors P(10)
!hpf$ distribute onto P :: A(block), B(cyclic), D(cyclic(5))
    interface
      subroutine iterate(X)
        real, dimension(100) :: X
!hpf$   ASSUME
!hpf$     processors Px(10)
!hpf$     ASSUME
!hpf$       distribute X(block) onto Px
!hpf$     OR
!hpf$       distribute X(cyclic) onto Px
!hpf$     END ASSUME
!hpf$   OR
          ! X not distributed!
!hpf$   END ASSUME
      end subroutine
    end interface
    call iterate(A) ! block
    call iterate(B) ! cyclic
    call iterate(C) ! not distributed
    call iterate(D) ! compiler chosen version (=> remapping)
  end program

  subroutine iterate(X)
    real, dimension(100) :: X, TMP
!hpf$ ASSUME
!hpf$   align with X :: TMP
!hpf$   processors P(10)
!hpf$   ASSUME
!hpf$     distribute X(block) onto P
!hpf$   OR
!hpf$     distribute X(cyclic) onto P
!hpf$   END ASSUME
!hpf$ OR
        ! X not distributed!
!hpf$ END ASSUME
    integer, save :: count
    count = count + 1
    TMP = X*X ! some computation...
    X = 1 + X + TMP + TMP*TMP
  end subroutine
```

Figure 3: Example with assume

where i is the instance number of the assumption set. This would lead to the three versions depicted in Figure 4. These three subroutines are standard HPF subroutines.

Handling of the `SAVE` attribute

In order to stick to the `SAVE` semantics the cloned versions must share the same address space for these data. The rewriting process can put them in some special `COMMON` (or `MODULE`) especially created for sharing data between cloned versions. Constraint 2 allows such a special area to be created as depicted in Figure 4 for saved variable `count`.

Compilation of the caller

From the caller point of view, the very same cloning must be applied on the assume part specified in the interface. The very same naming convention can be applied, because due to Constraint 1 the interface is in the same order as in the callee declaration. The caller must also switch the call sites to the right version, depending on the mapping of the array. This leads to the standard HPF program depicted in Figure 5.

Handling of `INHERIT`

It may happen that the actual argument mapping of an array is unknown from the caller, due to `inherit` specified on one of its own dummy arguments. In such cases the rewriting must mimic the decision process usually handled by the compiler, as depicted in Figure 6: array `Y` is unknown to the compiler, hence the generated code dynamically checks for the actual mapping in order to call the best version of `iterate`.

```
  subroutine iterate_1(X)
    real, dimension(100) :: X, TMP
!hpf$ align with X :: TMP
!hpf$ processors P(10)
!hpf$ distribute X (block) onto P
    common /iterate_save/ count
    integer count
    count = count + 1
    TMP = X*X ! some computation...
    X = 1 + X + TMP + TMP*TMP
  end subroutine

  subroutine iterate_2(X)
    real, dimension(100) :: X, TMP
!hpf$ align with X :: TMP
!hpf$ processors P(10)
!hpf$ distribute X (cyclic) onto P
    common /iterate_save/ count
    integer count
    count = count + 1
    TMP = X*X ! some computation...
    X = 1 + X + TMP + TMP*TMP
  end subroutine

  subroutine iterate_3(X)
    real, dimension(100) :: X, TMP
    common /iterate_save/ count
    integer count
    count = count + 1
    TMP = X*X ! some computation...
    X = 1 + X + TMP + TMP*TMP
  end subroutine
```

Figure 4: Cloned callee versions for example in Figure 3

```
  program assume
    real, dimension (100) :: A, B, C, D
!hpf$ processors P(10)
!hpf$ distribute onto P :: A(block), B(cyclic), D(cyclic(5))
    interface
      subroutine iterate_1(X)
        real, dimension(100) :: X
!hpf$   processors Px(10)
!hpf$   distribute X (block) onto Px
      end subroutine
      subroutine iterate_2(X)
        real, dimension(100) :: X
!hpf$   processors Px(10)
!hpf$   distribute X (cyclic) onto Px
      end subroutine
      subroutine iterate_3(X)
        real, dimension(100) :: X
      end subroutine
    end interface
    call iterate_1(A)
    call iterate_2(B)
    call iterate_3(C)
    call iterate_1(D) ! compiler chosen version (=> remapping)
  end program
```

Figure 5: Caller version after rewriting

```
  subroutine inherited(Y)
    real, dimension(100) :: Y
!hpf$ inherit Y
!hpf$ distribute * onto * :: Y
    ! iterate interface declaration...
    call iterate(Y)
  end subroutine

  subroutine inherited(Y)
    real, dimension(100) :: Y
!hpf$ inherit Y
!hpf$ distribute * onto * :: Y
    ! iterate cloned interface declaration...
    if (Y is distributed block) then
      call iterate_1(Y) ! no remapping
    elif (Y is distributed cyclic)
      call iterate_2(Y) ! no remapping
    elif (Y is not distributed)
      call iterate_3(Y) ! no remapping
    else
      call iterate_1(Y) ! remapping required
    end if
  end subroutine
```

Figure 6: Inherited unknown mapping management

5 Conclusion

The assume directive proposal for specifying several mappings at subroutine boundaries has important advantages, with respect to its expressiveness, its implementability and its simplicity:

- The full expressiveness of HPF mappings is used.
- Different mappings that depend on the reaching mappings may be specified for local arrays.
- Prescriptive mappings can be used as a default case: indeed the compiler can chose it as the version to call, whatever the incoming distribution, but at price of a remapping.
- It can be rewritten simply into a standard HPF program; thus it serves software engineering quality purposes without many requirements from the compiler technology.
- Hand-optimized library versions with different mapping assumptions can also benefit from this directive, by providing a unified interface to the function. For example, if `matmul` is available as library functions optimized for `(block,block)` (version 1), `(*,block)` (version 2) and *no* (version 3) distributions then the following interface would match these functions:

```
   interface
     subroutine matmul(A,B,C) ! computes A = B x C
       real, dimension(:,:) :: A, B, C
!hpf$ ASSUME
!hpf$   align with A :: B, C
!hpf$   distribute A(block,block)
!hpf$ OR
!hpf$   align with A :: B, C
!hpf$   distribute A(*,block)
!hpf$ OR
!hpf$ END ASSUME
     end subroutine
   end interface
```

- It provides a nice alternative to most uses of `inherit` in any real world program that is expected to be *high performance*, thus it should be considered for the core language. (`inherit` could be moved to approved extensions?)

Users should be warned that specifying an *assume-directive* can lead to high compilation costs due to the possible cloning performed by the compiler.

6 Open questions

- Some `empty` or `nothing` or `void` directive could be allowed so as to make it clear that *no* assumption is an assumption within an *assume-directive* when no directives are specified for a case.

```
!hpf$ ASSUME
!hpf$   align with T :: A, B
!hpf$   distribute T(block)
!hpf$ OR
!hpf$   nothing
!hpf$ END ASSUME
```

- consider another keyword instead of `ASSUME`?
- consider another keyword instead of `OR`: `ELSE`, `|` or whatever?
- Derived type mappings? no *assume-directive* within a type declaration, but may be allowed for variable instances.

```
      subroutine sub(a)
      type mmx
        real det, array(10000,10)
      end type mmx
      type(mmx) :: a
!hpf$ ASSUME
!hpf$   distribute (block,*) :: a%array
!hpf$ OR
!hpf$   distribute (cyclic,*) :: a%array
!hpf$ END ASSUME
      ...
      end subroutine
```

7 Journal of development

We thought of a shorhand syntax for ASSUME, to be specified directly within a directive:

7.1 Shorthand syntax

A shorthand syntax is also suggested to handle simple cases. It applies to any *specification-directive*. In such a directive, the `[`, `|` and `]` markers specify directly within the directive different assumptions, as depicted in Figure 7.

```
begin [ one | two | three ] end
```

would be strictly equivalent to:

```
ASSUME
  begin one end
OR
  begin two end
OR
  begin three end
END ASSUME
```

Figure 7: `[` and `]` shorthands for assume

Here are some examples of shorthand syntax specifications:

```
!hpf$ template T(100)
!hpf$ align with T( [ i | * ] ) :: A(i)
!hpf$ processors P [ (4) | (8) ]
!hpf$ distribute ( [ block | cyclic ] ) onto P :: T
```

Note 7 *The suggested shorthand syntax expressiveness is quite restricted compared to the full syntax. Also it implies implicitely the cross product of specifications if several are used.*

- `[`, `|` and `]` characters might cause interaction problems for some national character sets. We might think of `{` and `}` in their place? Or of trigraphs, such as `??(` and `??)`?
- Shorthand syntax issue: context free grammar? One might think of preprocessing the declarations to switch from the shorthand syntax to the full syntax, before any lexical analysis? Thus it would not be an issue. But is this preprocessing desirable? what if mandatory?

The syntax is dropped to avoid a mandatory preprocessing.