ISO/IEC JTC1/SC22/WG5 **N1186**

03 June 1996

High Performance Computing with Fortran 90 Qualifiers and Attributes

In context with optimizing Fortran 90 code it would be very helpful to have a selection of qualifiers as attributes or statements which could be ignored by the compiler or could be understood as semantic hints of the user. These qualifiers give information on relations and nonrelations between data objects to enable optimizations and inlining. Compilers have more possibilities to optimize if more information on the dependencies is given. Very often the compiler cannot determine dependencies between data objects.

It is imaginable that all those qualifiers and attributes specified below could have the same beginning like QUALIFY_ or CH_ to distinguish them from other Fortran attributes.

In the following a list of qualifiers and attributes is suggested, including

- I. *Injective* Qualifier for (Index) Arrays
- II. Restrictions for scalar and array pointers and their relations
- III. Keyword to allow inlining for trivial user defined procedures
- IV. Keyword to simplify inlining of procedures with pointer arguments
- V. Definition of independence of variables and arrays
- VI. Attribute to define a variable to be local on one processor on parallel systems
- VII. User control of copying array arguments
- VIII. Qualifier 'noloop' for (recursive) data types

This list could always be completed by others by the time. It also could be completed by companies defining their own qualifiers as e.g. CH_SGI_LOCAL or CH_CRAY_LOCAL, which are only recognized by the specific compilers.

Advantages of qualifiers and attributes :

- The qualifiers fit easily in the attribute list used for declaration of variables. Therefore it just fits right in the Fortran 90 declaration syntax.
- The qualifiers can also be ignored if they are meaningless on several systems
- Performance improvement

I. *Injective* Qualifier for Arrays

This qualifier is a hint for the compiler that an index array is injective to give better optimization possibilities. An index array is injective, if each of its values only turns up once in the whole array.

This optimization qualifier may be specified for each integer array

- as an attribute in the attribute list in the declaration
- as a simple statement right after the declaration part of a program or module.

Fortran is a programming language which is very strong in array handling. In many large applications index arrays have to be used to access a certain array element via indirect addressing. This is for example very important in FEM Codes. At compile time it cannot be known wether this index array is injective or not. To know this in advance would give the compiler the possibility to optimize, especially vectorize a loop.

Example I.1 :

```
do i=1,imax
  a(index(i)) = a(index(i)) + b(i)
enddo
```

This loop is not vectorized automatically because at compile time it cannot be known wether the index array `index` is injective or not.

Vectorization of loops is not only important for vector computers but also for super scalar machines, as each single processor has an architecture which supports handling vector code.

Helpful for optimization in context with vectorization of loops is a qualifier which gives information wether an index array is injective or not. Therefore one could think of an attribute e.g. `injective` which is simply added to the attribute list within the declaraion of the index array like in the example below:

Example I.2 :

```
integer,dimension(:),pointer,injective :: index
```

In the case that the array `index` is not injective, special algorithms can be used for the array handling.

If arrays have (rank $>$ 1), and not every dimension is to be defined injective, it could be thought of a statement called `injective` which takes as arguments the name of the array and the dimensions in which this array is injective.

Example I.3 :

```
integer,dimension(:,:,:,:),pointer :: index
...
injective(index,1,3,4)
```

In the example above (I.3) index is declared to be an integer array pointer of rank 4. The `injective` statement declares dimensions 1,3, and 4 of the array `index` to be injective.

II. Restrictions for Scalar and Array Pointers

This restriction in form of a qualifier gives the compiler a hint if two pointers are related only to different targets. Knowing this the compiler has better optimization possibilities.

This optimization qualifier may be specified for each scalar and array pointer pair

- as an attribute in the attribute list in the declaration part
- as a simple statement right after the declaration part of a program or module.

For optimization of statements including pointer components it is very useful to have a possibility to specify a pointer and a different target to be not related. Therefore the compiler knows that there cannot turn up any pointer dependencies between these two objects.

Relation information between data objects is very important for vectorizing code. Vectorization is still (and will always be) a very important optimization possibility not only on vector processors, due to the single processor behaviour in parallel and super scalar systems.

Fortran 77 - Example II.1 :

```
      subroutine add(a,b,c,nmax)

      real a(*),b(*),c(*)

      do 10 n=1,nmax
        a(i) = b(i) + c(i)
 10   continue

      return
      end

c=====================================

      program main

      real x(1000)

      do 20 i=1, ...
        x(i) = ...
 20   continue

      i1 = 2
      i2 = 1
      i3 = 200

c     not conforming to the standard
      call add(x(i1),x(i2),x(i3), 100)
      ...
      end
```

- The example above does not conform to the standard. In Fortran 77 dependencies between the data objects in the external subroutine `add` are not expected. With probably each compiler this example would produce wrong results. But this kind of coding (Example II.1) is typical and very common e.g. in developing Finite Element Code in Fortran 77. It also is the only possibility to ensure dynamical behaviour.
- In the C language, in a subroutine like the subroutine `add` dependencies between left and right hand side of the assignment in the loop always have to be expected. The compiler has the only chance to vectorize the loop by checking during runtime, wether the address ranges of the objects are overlapping or not. This is an expensive procedure.

- With Fortran 90 pointer any dependency has to be expected like in C. Therefore it would be very helpful for the user to have a possibility to tell the compiler somehow, that no dependencies are allowed to turn up between left and right hand side of the assignment.

Therefore it could be thought of adding qualifiers in form of either attributes or simple statements to the code. Using a qualifier named 'nonrelated', which means, that the specified data objects are not related and therefore the loop can be vectorized, the subroutine 'add' could look like one of the examples below.

Example II.2 :

```
subroutine add(a,b,c,nmax)

  real,dimension(:),pointer,nonrelated  :: a,b,c
  integer                               :: nmax

  do  ii=1,nmax
    a(ii) = b(ii) + c(ii)
  enddo

end subroutine add

!* alternative

subroutine add(a,b,c,nmax)

  real,dimension(:),pointer     :: a,b,c
  integer                       :: nmax
  nonrelated                    :: a,b
  nonrelated                    :: a,c

  do  ii=1,nmax
    a(ii) = b(ii) + c(ii)
  enddo

end subroutine add
```

Example II.3 : Finite Element Codes

- In Finite Element codes grids are managed by lists of elements (e.g. triangles).

  Each triangle has three neighbour cells, related either through Fortran 90 pointers or through indices by their identification number. Each triangle consists of three edges and three nodes. Each edge consists of two nodes.

There are a few ways to manage the whole set of triangles.

1. One possibility to organize the required elements are index lists. This is the way the elements would have to be managed in Fortran 77.

   Advantages :

   - index sets need less memory that pointers on several machines
   - implicit numbering of objects

   Disadvantages :

   - not flexible for self adaptive problems
   - not very elegant

2. Another possibility to organize the required elements are Fortran 90 pointer.

   Advantages :

   - can easily be expanded or reduced

   Disadvantages ::

   - poor efficiency in memory and performance due to the sizes of Fortran 90 pointer which might be very large, because a Fortran 90 pointer on some systems contains more information than just the address.

NAG	scalar pointer	4 bytes, same as C pointers
	array pointer	8 (12 for character arrays) bytes + 12 bytes * rank (Triplet)
Cray	scalar pointer	6 words (1 word = 8 bytes on CRAY)
	array pointer	6 words + 3 extra words * rank
IBM	scalar pointer (non character)	8 bytes
	scalar pointer (character)	12 bytes
	array pointer	20 bytes + 12 bytes * rank

   Variable access via pointers is more expensive then accessing a non pointer variable.

The example below shows a list of triangles organized in an array.

```
type status
  real            :: qq
end type status

type flux
  real            :: ff
end type flux

type koord
  real,dimension(2)       :: x
end type koord
```

```
type cell
  type(status)                    :: integral
  type(koord)                     :: sp
  real,dimension(3)               :: orient
  integer,dimension(3)            :: node
  integer,dimension(3)            :: side
  integer,dimension(3)            :: cell
endtype cell

type side
  type(flux)      :: flx
  type(koord)     :: df
  integer,dimension(2)    :: cell
  integer,dimension(2)    :: node
endtype side

type node
  type(koord)     :: xx
  type(koord)     :: vv
endtype node
```

From the logical side in the example above it is clear, that the three sides of the triangle are not related in the sense that one side could turn up twice. The same is valid for the nodes within one triangle. A triangle is built up by three different nodes. To give the compiler the possibility for better optimization, the derived type for e.g. a cell could look like the following.

```
type cell
  type(status)                    :: integral
  type(koord)                     :: sp
  real,dimension(3)               :: orient
  type(node),pointer,nonrelated   :: node1,node2,node3
  type(side),pointer,nonrelated   :: side1,side2,side3
  type(cell),pointer,nonrelated   :: cell1,cell2,cell3
endtype cell

!* or

type cell
  type(status)                    :: integral
  type(koord)                     :: sp
  real,dimension(3)               :: orient
  type(node),pointer              :: node1,node2,node3
  type(side),pointer              :: side1,side2,side3
  type(cell),pointer              :: cell1,cell2,cell3
  nonrelated                      :: node1,node2
  nonrelated                      :: cell1,cell2,cell3
endtype cell
```

III. Keyword to allow Inlining for trivial user defined procedures

The keyword gives the compiler the hint that a function or subroutine is trivial and without side effects and can be inlined without any difficulties.

This can result in performance improvement by avoiding function and subroutine calls if the procedures are inlined by the compiler. It is users choice to define a procedure to be trivial and give it the `inline` keyword which allows the compiler to inline the procedure without restrictions.

Object oriented (similar to component oriented) programming style takes the fortune out of small encapsulated objects consisting on a data structure, methods, and interfaces which guarantee the accessibility to objects from outside. As those objects are typically small, many function and / or subroutine calls have to be made to access the root components of an object of a complex system. Many functions and subroutines have to be implemented simply to read the values of a component on a certain abstraction level. An example therefore is the component describing the dimension in row index direction of a matrix.

Example III.1 :

```
module matrix_info_module

  type matrix_info_type
    type(strng) :: name
    integer     :: row_index_dim, col_index_dim
  end type matrix_info_type
      ...

  interface get_row_dim
    module procedure get_row_dim_matrix_info
  end interface
      ...

  function get_row_dim_matrix_info(info) result(dim)

    type(matrix_info_module)  :: info
    integer                   :: dim

    dim = info%row_index_dim

  end function get_row_dim_matrix_info
      ...

end module matrix_info_module

!*-----------------------------------------------

module square_matrix_mdoule

  use matrix_info_module
```

```
  type square_matrix_type
    type(matrix_info_type)      :: info
    real,dimension(:,:),pointer :: values
  end type square_matrix_type

  interface get_row_dim
    module procedure get_row_dim_square_mat
  end interface
      ...

  function get_row_dim_square_mat(mat) result(dim)

    type(square_matrix_type)  :: mat
    integer                   :: dim

    dim = get_row_dim(mat%info)

  end function get_row_dim_square_mat
      ...
end module square_matrix_mdoule
```

In the example above (III.1) two functions have to be called just to determine the dimension of a matrix of `type(square_matrix_type)` in row index direction. This is absolutly in the sense of encapsulation, but of course this gives a hint that object oriented programming might cause cascades of function calls which might be performance decreasing, even more if inheritance is not supported by the programming language. In this case it should be possible for a user to specify trivial functions to be inlined to improve performance.

This should be possible even if the function / subroutine is declared to be private. The accessibility restrictions of the procedure are still valid, but the code is visible after the inlining step. If the `private` statement is also used to hide the implementation of a certain function or subroutine completely from the user, it should also be possible to declare this function never to be inlined.

Therefore one could think of statements like `inline` or `noinline` which can be placed as a first statement in the function or subroutine. On preprocessor level, these function / subroutines could simply be inlined. Example III.2 shows a function which can easily be inlined and therefore has the `inline` statement as a first statement. Example III.3 shows the specification of the function `solve_linear_system` which is not to be inlined, because it might contain a rather complicated well optimized code to solve linear systems which the developer does not want to be visible to users.

Example III.2 :

```
module matrix_info_module
      ...
  function get_row_dim_matrix_info(info) result(dim)

    inline

    type(matrix_info_module)  :: info
    integer                   :: dim

    dim = info%row_index_dim

  end function get_row_dim_matrix_info
      ...
end module matrix_info_module
```

Example III.3 :

```
module linear_solvers

  use ...
  ...

  function solve_linear_system(mat,vec) result(res_vec)

    noinline

    type(square_matrix_type)  :: mat
    type(vector_type)         :: vec, res_vec
       ...
  end function solve_linear_system

end module linear_solvers
```

IV. Keyword to Simplify Inlining

Qualifier specified by (II.) used in context with overloaded operators, to give the user the possiblity to specify left and right hand side (operators and result value) to be not related. This also gives the compiler a possibiliy to inline the code.

Overloaded operators in Fortran 90 might cause difficulties if inlined, because it cannot be known if left and right hand side (operators and result value) are related somehow or

not. To simplify the inlining choice one could think of additional qualifiers like one forbidding recursions between left and right side. These qualifiers of course mean restrictions somehow, but enable a simpler way of inlining.

V. Separation of Variables and Arrays

Define arrays and variables to be independent from each other and to be used independently to improve performance. This is important for cache coherent parallel systems which can take a fortune out of this by storing data in different cache lines, but is also important helpful in context with all future hardware architectures.

To improve performance one could think of a qualifier for arrays and variables to declare them to be independent. This could cause the arrays to be stored in different cache lines which will improve performance

Examples V.1 :

```
  real,dimension(:),pointer,independent :: aa,bb

!* or ::

  real,dimension(:),pointer :: cc
  real,dimension(nmax)      :: dd

  independent(cc,dd)
```

VI. Attribute for Local Variables on Parallel Systems

Declaration of a variable or an array to be local on one process in parallel systems. This means that no other process running on a different CPU accesses this object.

In HPF any object which is not distributed is local to the process. It will also be necessary to define a kind of global objects which can be accessed by multiple processes. Having an attribute gives compilers the possibility for better optimization if it can be distinguished between global and local variables in this context.

Convex already has some kind of virtual memory classes to be able to distinguish between private (thread-private, node-private) and shared (near-shared, far-shared, and block-shared) variables.

The qualifier may be specified either as an attribute in the attribute list of a variable, or as a simple statement.

It could be thought of an additional attribute `nonshared` or `local` which declares a variable or an array to be local on one processor.

Example VI.1 :

```
real,dimension(:),pointer,nonshared  :: aa
real,dimension(nmax)                 :: bb

nonshared(bb)
```

VII. User Control for Copying of Actual Arguments

The Fortran 90 Array Syntax allows a user to pass array sections to procedures. Passing array sections results in copying the argument. As a compiler cannot know what kind of array is passed to the procedure (array or array section), it might always result in copies. This of course is very much performance decreasing, especially if large arrays are passed.

There should be a possibility for a user to control the way array arguments are passed to procedures via both explicit and implicit interfaces. This results in performance improvement by avoiding copies where not necessary.

Explicit Interfaces :

Using explicit interfaces, copying of arguments should be assessable by the user himself via a qualifier, while the default value without qualifier is the usual behaviour as we have it now. The qualifier could be added to the attribute list of the argument declaration part of the procedure.

Example VII.1.

```
subroutine sub(a,b)
  real,dimension(:),nocopy  :: a,b
          ...
end subroutine sub
```

Implicit Interfaces :

Implicit interfaces do not cause any problems if already compiled Fortran 77 programs (e.g. Libraries) (compiled with a Fortran 77 compiler) are used, because in Fortran 77, array sections are not known.

External Fortran 90 procedures with arguments having the pointer attribute or with derived type arguments require explicit interfaces anyway to be possible. Therefore the argument passing again will be managed via the 'nocopy' attribute within the interface.

Fortran 90 subprograms having arrays as dummy arguments can be called via implicit interfaces. But as Fortran 90 allows passing array sections, the compiler cannot know wether the argument has to be copied or not.

Sometimes a user has got to use implicit interfaces in his program for some reason. This causes copying of the array arguments, wether array sections are passed or not. And copying of arguments is expensive.

Therefore even for implicit interfaces it has to be controllable by the user to specify a procedure to be called by reference.

This can be done either by prohibiting passing array sections to procedures (If required, the user has to make an explicit copy of the argument himself) or by a statement like CALL_BY_REFERENCE which can be specified for several Fortran 90 procedures and will simply be ignored for others.

Example VII.2. :

A very important example for the problem is the Fortran 90 interface for MPI.

In the Fortran 90 interface for MPI the type of one argument very often is users choice. For example if sending a message, this message does not have to be of an instrinsic type, but can also be of user defined type. The message to be sent is passed to the sending routine as an argument. These user defined types of course are not known at compile time of the MPI Library. Therefore no explicit interfaces can be used for the MPI interface because argument passing via explicit interfaces results in argument type checking of actual and dummy arguments. The only restriction is a sequence of the object components in the memory to make a call by reference possible and avoid copies. Implicit interfaces do not bother on the types of the arguments. But as soon as arrays are passed, it is possible to pass array sections, therefore passing arrays will probably always result in copies.

MPI works with references of objects and therefor provides a procedure to determine the address of an object. If an array `a` is passed to a subroutine `sub1` and a copy of the argument is made, another process which wants to work on `a` via its reference will not get the original reference of `a` but of its copy within the subroutine `sub1` if it is called in some nested context within `sub1`.

This shows another important reason for the need of a user definable call by reference,

apart from performance advantages.

VIII. Qualifier `noloop` for data types

The `noloop` qualifier gives the compiler the hint, that only linear lists without recursions are built up with the data type containing the qualifier.

The qualifier can be used e.g. if recursive data types are defined which are to be used for creating trees or linked lists without recursions.

Example VIII.1 :

```
type mytype
  integer                     :: counter
  type(mytype),pointer,noloop :: next
end type mytype
```