Fortran features in OpenMP

Kelvin Li (kli@ca.ibm.com)

IBM Canada Lab

Content

- associate construct
- mapping Fortran pointers and allocatables
- runtime routines

It is not …

- An exhaustive list
- A how-to-write-best OpenMP program in Fortran material

It is …

- Some minor points about some Fortran specific behavior in the OpenMP spec
- Hope that it helps in writing or porting Fortran code to OpenMP
- Get feedback and comments about the current features

Associate construct

- “The ASSOCIATE construct associates named entities with expressions or variables during the execution of its block. These named construct entities are associating entities. The names are associate names.”
- added in F2003
- supported in OMP4.0
- useful in associating with multiple occurrence of any complex expression in a block of code

```
associate (associate-name => selector)
  ...
end associate
```

Associate construct

- “Execution of an ASSOCIATE construct causes evaluation of every expression within every selector that is a variable designator and evaluation of every other selector, followed by execution of its block. During execution of that block each associate name identifies an entity which is associated with the corresponding selector. ...” (F2018)
- can associate with an expression or a variable

```
associate (z => xdt%x(i))
  ... = z

  z = z + const
end associate
```

```
associate (z => xdt%x(i) + xdt%y(i-1) + xdt%z(i+1))
  ... = z

  ...

  ... = z
end associate
```

Associate construct

Not quite the same as reference type in C++

- the binding cannot be changed within a given scope
- cannot bind to an expression

```fortran
associate (z => xdt%x(i))
  ... = z

  z = z + const
end associate
```

```cpp
{
  float &z = xdt.x[i];
  ... = z;

  z = z + const;
}
```

```fortran
associate (z => xdt%x(i) + xdt%y(i-1) + xdt%z(i+1))
  ... = z
  xdt%x(i) = 1
  ... = z
end associate
```

```cpp
float &z = xdt.x[i] + xdt.y[i-1] + xdt.z[i+1];  // invalid
```

Associate construct

What OpenMP spec says about the associate construct?

- have a *predetermined* data-sharing attribute:
  "An associate name that may appear in a variable definition context is shared if its association occurs outside of the construct and otherwise it has the same data-sharing attribute as the selector with which it is associated." (OMP5.2)
- when the association is established is important
  - before encountering the OMP construct, OR
  - during the execution of the OMP construct

```
associate (z => x)

  !$omp parallel private(z) ! invalid
    z = z + 1
  !$omp end parallel

end associate
```

z has the predetermined data-sharing attribute of shared and it is not allowed to appear in any data-sharing attribute clause.

```
associate (z => x + y)

  !$omp parallel private(z) ! invalid
    t = z + 1
  !$omp end parallel

end associate
```

z cannot appear in a variable definition context hence "x + y" is not allowed to be a *variable list item*.

Associate construct

What OpenMP spec says about the associate construct?

- “A privatized list item may be a selector of an ASSOCIATE or SELECT TYPE construct. If the construct association is established prior to a parallel region, the association between the associate name and the original list item will be retained in the region.” (OMP5.2)

```
!$omp parallel private(x)

  associate(z => x)
    z = z + 1
  end associate

!$omp end parallel
```

A construct association is established between z and private x.
Any reference of z in the associate construct is to the private x.

```
!$omp parallel private(y)

  associate(z => x + y)
    t = z + 1
  end associate

!$omp end parallel
```

A construct association is established between z and private x.
Any reference of z inside the associate construct is to
(x + private y).

Associate construct

How about using it in offloading?

```
associate (z => dt%a%x(1:5))

  !$omp target map(z)
    z = z + 1
  !$omp end target

end associate
```

dt%a%x can appear in a variable definition context, it is allowed on the map clause (mapping z is as if mapping dt%a%x)

```
associate (z => dt%a%x(1) + dt%a%y(1))

  !$omp target map(z) ! invalid
    z = z + 1
  !$omp end parallel

end associate
```

z is not allowed on the map clause as it cannot appear in a variable definition context

Mapping Fortran variables

- mapping Fortran pointer and allocatable is always intriguing (or confusing)
- some behaviors are different from C/C++ due to different language characteristics
- handling pointers and allocatables usually has more stuff going on under the hook
    - an implementation usually has a descriptor (a.k.a. a dope vector) to represent the allocatable, pointer variables and other entities – to store bounds, rank, extent, and other necessary information
    - but the base language standard does not say anything about descriptors prior to F2018
    - makes the description in the spec difficult

Mapping Fortran variables

- mapping allocatable variables – the easier one!
- mapping a derived type with allocatable components - it is a DEEP copy
- alternatives if only a few allocatable components are needed
  - map components individually – map what is needed
  - use declare mapper

```
type dt
  real, allocatable :: a(:)
  real, allocatable :: b(:)
  ...
  real, allocatable :: z(:)
end type
type(dt) :: xdt

! all components are allocated
!$omp target map(xdt)
  ...
!$omp end target map
```

```
type dt
  real, allocatable :: a(:)
  real, allocatable :: b(:)
  ...
  real, allocatable :: z(:)
end type
type(dt) :: xdt

! all components are allocated
!$omp target map(xdt%a, xdt%b)
  ! only xdt%a and xdt%b are needed
  ...
!$omp end target map
```

```
type dt
  real, allocatable :: a(:)
  real, allocatable :: b(:)
  ...
  real, allocatable :: z(:)
end type
type(dt) :: xdt
!$omp declare mapper(myMap : type(dt)::t) map(t%a, t%b)

! all components are allocated
!$omp target map(mapper(myMap): xdt)
  ! only xdt%a and xdt%b are needed
  ...
!$omp end target map
```

Mapping Fortran variables

- be careful if you map before allocate
- “If the allocation status of an original list item that has the ALLOCATABLE attribute is changed while a corresponding list item is present in the device data environment, the allocation status of the corresponding list item is unspecified until the list item is again mapped with an **always** modifier on entry to a **target**, **target data** or **target enter data** region.”

```
!$omp target data map(x)
  allocate(x(10))

!$omp target map(always, tofrom: x)
  x(1:N) = ...
!$omp end target

!$om end target data
```

allocation occurs on the host but not on the device

```
module m
  real, allocatable :: x(:)
!$omp declare target enter(x)
end module

  use m
  allocate(x(10))

!$omp target map(always, tofrom: x)
  x(1:N) = ...
!$omp end target
```

Mapping Fortran pointer

- involve at least two objects (i.e. pointer and pointer target)
- “For **map** clauses on map-entering constructs, if any list item has a base pointer for which a corresponding pointer exists in the data environment upon entry to the region and either a new list item or the corresponding pointer is created in the device data environment on entry to the region, then:

  The corresponding pointer variable is associated with a pointer target that has the same rank and bounds as the pointer target of the original pointer, such that the corresponding list item can be accessed through the pointer in a **target** region.”

```
  real, pointer :: p(:)
  real, target :: t(10)

!$omp target enter data map(t)
!$omp target map(p)

  p(1::2) = ...
  p(2::2) = ...

!$omp end target
```

Mapping Fortran pointer

- derived types with pointer components are NOT deep copy
- need to map the component individually

```
type dt
  real, pointer :: p(:)
  ...
end type
type(dt) :: xdt

  xdt%p => t(:)

!$omp target map(xdt)

  ... = associated(xdt%p)  ! F
!$omp end target
```

```
type dt
  real, pointer :: p(:)
  ...
end type
type(dt) :: xdt

  xdt%p => t(:)

!$omp target map(xdt, xdt%p)

  ... = associated(xdt%p)  ! T
!$omp end target
```

Fortran interface for runtime routines

- quite a few bind(c) routines

```
omp_target_alloc
omp_target_free
omp_target_is_present
omp_target_is_accessible
omp_target_memcpy
omp_target_memcpy_rect
omp_target_memcpy_async
omp_target_memcpy_rect_async
omp_target_associate_ptr
omp_target_disassociate_ptr
omp_get_mapped_ptr
```

```
omp_init_allocator
omp_destroy_allocator
omp_set_default_allocator
omp_get_default_allocator
omp_alloc
omp_aligned_alloc
omp_free
omp_calloc
omp_aligned_calloc
omp_realloc
```

Fortran interface for runtime routines

- for the bind(c) routines
- advantages
  - get around some difficulties to have a Fortran entity to interop with the C counterpart (e.g. handling `void*`)
  - allow implementation to directly call the runtime routines (likely written in C or C++ or both)
- disadvantages
  - the C interop data type used in the routines may proliferate to the other part of the program (if portability is a concern)
  - some extra steps may need to take before calling the routines

Fortran interface for runtime routines

- TKR (type, kind and rank)
  - type compatible
  - same kind type parameter
  - matching rank
- for example, omp_target_alloc

```
interface
  type(c_ptr) function omp_target_alloc(size, device_num) bind(c)
  use, intrinsic :: iso_c_binding, only : c_ptr, c_size_t, c_int
  integer(c_size_t), value :: size
  integer(c_int), value :: device_num
  end function
end function
!
type(c_ptr) :: cptr
integer(c_size_t) :: sz
integer :: dev

cptr = omp_target_alloc(sz, dev)
```

The kind type parameter of the default integer is assumed to be the same as integer(c_int).

This call may result in incompatible interface with some implementations.

Fortran interface for runtime routines

- TKR (type, kind and rank)
  - type compatible
  - same kind type parameter
  - matching rank
- for example, omp_target_alloc

```
interface
  type(c_ptr) function omp_target_alloc(size, device_num) bind(c)
  use, intrinsic :: iso_c_binding, only : c_ptr, c_size_t, c_int
  integer(c_size_t), value :: size
  integer(c_int), value :: device_num
  end function
end function
!
type(c_ptr) :: cptr
integer(c_size_t) :: sz
integer :: dev

cptr = omp_target_alloc(sz, dev)
```

```
type(c_ptr) :: cptr
integer(c_size_t) :: sz
integer :: dev
integer(c_int) :: c_dev

c_dev = dev
cptr = omp_target_alloc(sz, c_dev)
```

```
type(c_ptr) :: cptr
integer(c_size_t) :: sz
!!! integer :: dev
integer(c_int) :: c_dev

cptr = omp_target_alloc(sz, int(c_dev,kind=c_int))
```

Fortran interface for runtime routines

- some extra steps may need get access to things that are returned from the routine
    - may need to “translate / convert” it to a Fortran entity etc.
- for example, omp_target_alloc

```
type(c_ptr) :: cptr
integer(c_size_t) :: sz
integer(c_int) :: dev
real :: fptr(:,:)

cptr = omp_target_alloc(sz, dev)

!$omp target is_device_ptr(cptr)
  call c_f_pointer(cptr, fptr, [N,4])

  do i=1,N
    fptr(i,:) = ...
  enddo
!$omp end target
```

Need to associate a Fortran data pointer with the target of the C pointer by calling c_f_pointer procedure.

Fortran interface for runtime routines

- using C interoperability feature to define Fortran interfaces
  - does it add extra burden to users porting code to OpenMP?
  - any feedback is welcome

Fortran interface for runtime routines

- some missing pieces
- not all the OMP runtime routines have Fortran interface
  - OMPT and OMPD routines – probably the chance of the tooling written in Fortran is quite low
  - interoperability routines – an open issue in the language committee
    - is it needed?
    - any input is welcome
  - some future features
- Will the F2018 interoperability features help (e.g. assumed-type `type(*)`, assumed-rank `dimension(..)` etc.)?


Thank you!


Feedback
Welcome!