Use of f90 user-defined types and operators in high-level code

V. Balaji
SGI/GFDL

GFDL Princeton University
14 March 2000

Overview

- Parallel shallow water model example
- Derived types, user-defined assignment and operators
- Treatment of halo regions
- f90 issues (pointers, etc)
- Comparison with C++

Parallel shallow water model

```
program shallow_water
  use mpp_domains_mod
  use distributed_grids
  type(scalar2D) :: eta(0:1)
  type(hvector2D) :: utmp, u, forcing
  real :: dt, h, g
  integer tau=0, taup1=1
...
  f2 = 1./(1.+dt*dt*f*f)
  do l = 1,nt
     eta(taup1) = eta(tau) - (dt*h)*div(u)
     utmp = u - (dt*g)*grad(eta(tau)) + (dt*f)*kcross(u) + dt*forcing
     u = f2*( utmp + (dt*f)*kcross(utmp) )
     tau   = 1 - tau
     taup1 = 1 - taup1
  end do
end program shallow_water
```

- Runs and reproduces answers on t90, t3e, SGI.
- No parallel calls.
- Memory scaling (except for halo region overhead).
- 400 Mflops, 800 Mmops, on t90 125×125.
- 80% scaling on 5×5 PEs on t3e.
- MOM4 ($3^\circ \times 3^\circ$) free surface comparison: 77 vs 52 sec on 2 PEs.

Modules

```
module distributed_grids
  use mpp_domains_mod
  implicit none
  private
  type, public :: scalar2D
     real, pointer :: data(:,:)
     integer :: is, ie, js, je
  end type scalar2D
  type, public :: hvector2D
     type(scalar2D) :: x, y
     integer :: is, ie, js, je
  end type hvector2D
```

- Modules provide *protected namespaces* and *data-hiding*.
- User-defined types provide *data encapsulation*.
- `use` statements provide *inheritance*.

```
type, public :: scalar2D
   real, pointer :: data(:,:)
   integer :: is, ie, js, je
end type scalar2D
```

- Type component arrays in f90/95 must be *pointer* or *static*. This is being remedied in f2k. Allocatable type components will be available in cf90 3.5.

- `is, ie, js, je` contain the *active domain*.

Assignment of derived types

```
type(scalar2D) :: a, b, c
...
a = b
```

f90 provides an intrinsic assignment of derived types (“automatic inheritance”). However, there is a problem in that the standard specifies that pointers must be redirected by an assignment. Thus, certain constructs may not work as expected:

```
!interchange a and b
c = a
a = b
b = c
```

Also, this will begin to work as expected with allocatable components!

User-defined assignment

```
  interface assignment(=)
!copy
     module procedure copy_scalar2D_to_scalar2D
     module procedure copy_hvector2D_to_hvector2D
!assign arrays of various ranks to grid field types
!scalar2D
     module procedure assign_0D_to_scalar2D
     module procedure assign_2D_to_scalar2D
!hvector
     module procedure assign_2D_to_hvector2D
  end interface
```

f90 requires the procedure to be a *subroutine* with exactly two arguments: an `intent(inout)` LHS and an `intent(in)` RHS.

User-defined operators

```
use distributed_grids
type(scalar2D) :: a, b, c
...
c = a + b
...
module distributed_grids
  interface operator(+)
     module procedure add_scalar2D
     module procedure add_hvector2D
     module procedure add_scalar3D
     module procedure add_hvector3D
  end interface
```

f90 requires the procedure to be a *function* with exactly two arguments, both `intent(in)`.

add_scalar2D

```
    function add_scalar2D( a, b )
      type(scalar2D) :: add_scalar2D
      type(scalar2D), intent(in) :: a, b
      add_scalar2D%data => work2D(:,:,nbuf2)
!addition is done on valid domain
      add_scalar2D%is = max(a%is,b%is)
      add_scalar2D%ie = min(a%ie,b%ie)
      add_scalar2D%js = max(a%js,b%js)
      add_scalar2D%je = min(a%je,b%je)
!dir$$ IVDEP
      do j = add_scalar2D%js,add_scalar2D%je
         do i = add_scalar2D%is,add_scalar2D%ie
            work2D(i,j,nbuf2) = a%data(i,j) + b%data(i,j)
         end do
      end do
      nbuf2 = mod( nbuf2+1,nbufs )
      return
    end function add_scalar2D
```

add_scalar2D design issues: allocation

The function result is effectively `intent(out)`.

- Space can't be borrowed from the LHS, since you might have `c = a + b` or `d = (a + b) + c`.

- Allocating space for pointers is a) slow; b) potentially leaky.

```
real, pointer :: a(:)
allocate( a(100) )
...
a => b(1:100)
```

- Use of internal buffers seems to be the correct solution.

add_scalar2D design issues: allocation

```
    subroutine grid_domain_init
...
      allocate( work2D(isd:ied,jsd:jed,nbufs) )

    function add_scalar2D( a, b )
      type(scalar2D) :: add_scalar2D
      type(scalar2D), intent(in) :: a, b
      add_scalar2D%data => work2D(:,:,nbuf2)
...
      nbuf2 = mod( nbuf2+1, nbufs )
    end function add_scalar2D
```

`nbufs` must be greater than the length of your longest chain.

```
a = b + c + d + e + f ... !probably only requires 2 buffers
a = (b + c) + (((d + e) + f) + (g + h))
```

add_scalar2D design issues: aliasing

```
!dir$ IVDEP
      do j = add_scalar2D%js,add_scalar2D%je
         do i = add_scalar2D%is,add_scalar2D%ie
            work2D(i,j,nbuf2) = a%data(i,j) + b%data(i,j)
         end do
      end do
```

Since arguments are pointers, the compiler cannot know whether they point to the same or different memory. `IVDEP` provides a hint.

add_scalar2D design issues: active domains

The function result is effectively `intent(out)`.

```
!addition is done on active domain
      add_scalar2D%is = max(a%is,b%is)
      add_scalar2D%ie = min(a%ie,b%ie)
      add_scalar2D%js = max(a%js,b%js)
      add_scalar2D%je = min(a%je,b%je)
```

- All operators act on *active domain*, which includes all points in the data domain that contain valid data.

- Sum is done over intersection of active domains.

Inheritance

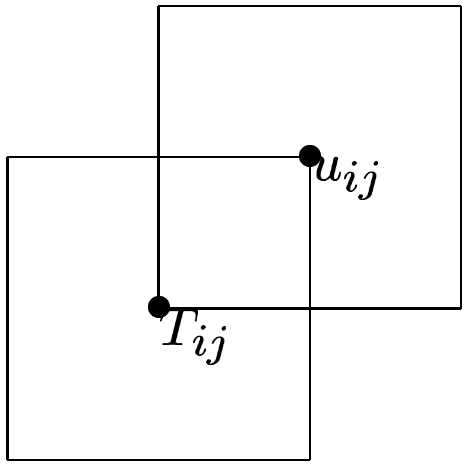
```
  type, public :: hvector2D
     type(scalar2D) :: x, y
     integer :: is, ie, js, je
  end type hvector2D
...
    function add_hvector2D( a, b )
      type(hvector2D) :: add_hvector2D
      type(hvector2D), intent(in) :: a, b
      add_hvector2D%x = a%x + b%x
      add_hvector2D%y = a%y + b%y
      add_hvector2D%is = add_hvector2D%x%is
      add_hvector2D%ie = add_hvector2D%x%ie
      add_hvector2D%js = add_hvector2D%x%js
      add_hvector2D%je = add_hvector2D%x%je
      return
    end function add_hvector2D
```

div and grad

$$\nabla \cdot \mathbf{u} = \delta_x(\overline{u}^y) + \delta_y(\overline{v}^x) \tag{1}$$

$$(\nabla T)_x = \delta_x(\overline{T}^y) \tag{2}$$

$$(\nabla T)_y = \delta_y(\overline{T}^x) \tag{3}$$



```
    function grad_scalar2D(scalar)
      type(hvector2D) :: grad_scalar2D
      type(scalar2D), intent(inout) :: scalar
...
      if( scalar%ie.LE.ie .OR. scalar%je.LE.je )then
          call mpp_update_domains( scalar%data, domain, EUPDATE+NUPDATE )
          scalar%ie = ied
          scalar%je = jed
      end if
      grad_scalar2D%is = scalar%is; grad_scalar2D%ie = scalar%ie - 1
      grad_scalar2D%js = scalar%js; grad_scalar2D%je = scalar%je - 1

!dir$ IVDEP
      do j = grad_scalar2D%js,grad_scalar2D%je
         do i = grad_scalar2D%is,grad_scalar2D%ie
            tmp1 = scalar%data(i+1,j+1) - scalar%data(i,j)
            tmp2 = scalar%data(i+1,j) - scalar%data(i,j+1)
            work2D(i,j,nbuf2) = gradx(i,j)*( tmp1 + tmp2 )
            work2D(i,j,nbufy) = grady(i,j)*( tmp1 - tmp2 )
         end do
      end do
```

Features of differencing operators

- Details of numerics are hidden from high-level code.
- Highly optimized numerical kernels without sacrificing readability.
- Extensible: can overload different algorithms as required or desired.
- Grid metrics are set once, at initialization.
- Update domains only as required, with no user intervention, including one-sided updates.
- Builtin use of *wide halos* for balancing computation with communication.

Wide halos

On a machine with a slow interconnect, we can choose to replace communication by redundant computation:

- Points in the active domain may be computed on more than one PE.
- Active domain is reduced until there are not enough points left to update the computational domain.
- Then update halos. This may only occur once every several timesteps.

```
call mpp_update_domains( ..., xhalo=1, yhalo=1 )
call mpp_update_domains( ..., xhalo=6, yhalo=6 )
```

Comparison with C++

“With the advent of f90, we finally have a compiler that runs as slow as C++.”

Features of f90 we use:

- Class libraries with objects and methods.
- Namespaces and data hiding.
- Inheritance.
- Polymorphism.

“f90 is C++ with fast computational kernels.”