Abstract parallel dynamical kernels for flexible climate models

V. Balaji
SGI/GFDL

ECMWF TeraComputing Workshop
16 November 2000

GFDL

GFDL is a climate modeling centre. The primary focus is the use of coupled climate models for simulations of climate variability and climate change on short and long time scales.

Current computing capability: Cray T90 24p, T3E 128p.

Future computing capability: $8 \times 128 + 2 \times 64$p Origin 3000.

GFDL models

- MOM: Modular Ocean Model.
- FMS: Flexible Modeling System.
- Hurricane model.
- HIM: isopycnal model.
- 2 non-hydrostatic atmospheric models.
- Older models: SKYHI, Supersource.

Modernization

- Parallelism without compromising vector performance.
- Modular design for interchangeable dynamical cores and physical parameterizations. Several dynamical cores are currently available.
- Distributed development model: many contributing authors. Use high-level abstract language features for encapsulation, polymorphism.

FMS: Flexible Modeling System

Dynamical cores:

- Atmosphere:
  - Hydrostatic spectral
  - Hydrostatic Arakawa B grid
  - Hydrostatic Arakawa C grid (*)
  - Non-hydrostatic Arakawa C grid (*)
- Ocean:
  - B grid
  - C grid (*)
  - Generalized vertical coordinate (*)

FMS: Physical processes

- Atmosphere:
  - Deep convection.
  - Shallow convection.
  - Moist processes.
  - Cloud mass flux.
  - Ozone, CFCs, greenhouse gases.
  - Radiation.
  - Turbulence.
  - Planetary boundary layer.
  - Land surface, ocean surface.

Climate models

Climate models solve the initial value problem of integrating forward in time the state of all the components of the planetary climate system. The underlying dynamics is the solution of the non-linear Navier-Stokes equation on a sphere. While the dynamics itself is the same for a wide variety of problems, resolutions and lengths of integration vary over several orders of magnitude of time and space scales. Efficient integration for different problems require different representations of the basic numerical kernels, which may also be a function of the underlying computer architecture on which the simulations are done.

Abstraction

Modern languages such as Fortran 90 and C++ offer the possibility of abstract representations of the the basic dynamical operators. These abstractions offer a large measure of flexibility in the dynamical operator code, without requiring large-scale rewriting for different problem sizes and architectures. The cost of this abstraction is a function of the maturity of the compiler as well as the language design.

Class libraries

Class libraries offer a clean, modular, extensible approach to building models using the components that most resemble the conceptual categories of the modeled system. In contrast to a traditional library, which provides a set of subroutines fulfilling certain needs, a class library defines a class of *objects* that you wish to work with, and the *methods* for those objects.

A well-kept secret is that *f90* modules allow one to build class libraries, having most of the useful features, but few of the current performance disadvantages of OO languages (C++, Java).

Overview

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

mpp_domains_mod: domain class library

Definition of *domain*:

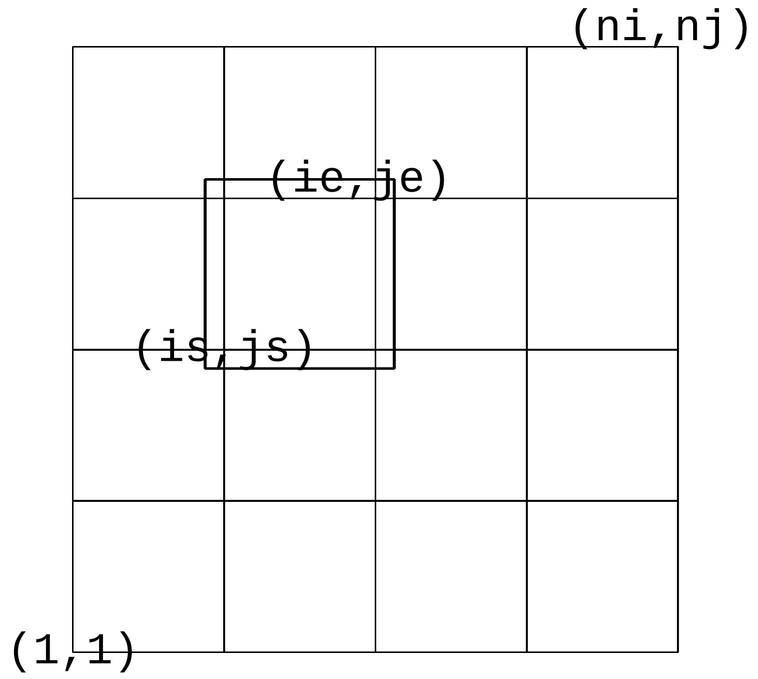
- *Global domain*: the entire model grid.
- *Compute domain*: set of points calculated by a PE.
- *Data domain*: set of points required by the computation (i.e including halo).

All the information required for domain-related operations are maintained in compact form in the *domain* types supplied by mpp_domains_mod. Complicated grids, such as the bipolar grid and the cubed sphere can be represented in this class, so long as they are logically rectilinear.

The *domain* type

```
type, public :: domain_axis_spec
   integer :: start_index, end_index, size, max_size
   logical :: is_global
end type domain_axis_spec
type, public :: domain1D
   type(domain_axis_spec) :: compute, data, global
   integer :: ndomains
   integer :: pe
   integer, dimension(:), pointer :: pelist
   type(domain1D), pointer :: prev, next
end type domain1D
```

```
!domaintypes of higher rank can be constructed from type domain1D
  type, public :: domain2D
     sequence
     type(domain1D) :: x
     type(domain1D) :: y
     integer :: pe
     type(domain2D), pointer :: west, east, south, north
  end type domain2D
```



mpp_domains_mod calls:

- `mpp_define_domains()`
- `mpp_update_domains()`

```
type(domain2D) :: domain(0:npes-1)
call mpp_define_domains( (/1,ni,1,nj/), domain, xhalo=2, yhalo=2 )
...
!allocate f(i,j) on data domain
!compute f(i,j) on compute domain
...
call mpp_update_domains( f, domain(pe) )
```

Parallel numerical kernels

$$\frac{\eta^{n+1}-\eta^{n}}{\Delta t} = -H(\nabla\cdot\mathbf{u})^{n} \quad (1)$$

$$\frac{\mathbf{u}^{n+1}-\mathbf{u}^{n}}{\Delta t} = -g(\nabla\eta)^{n+1} + f\mathbf{k}\times\left(\frac{\mathbf{u}^{n+1}+\mathbf{u}^{n}}{2}\right) + \mathbf{F} \quad (2)$$

```
program shallow_water
  type(scalar2D) :: eta(0:1)
  type(hvector2D) :: utmp, u, forcing
  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(taup1)) + (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, Beowulf.
- 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.
- Abstraction penalty about 20% on MOM 2p.
- Standard f90 (Cray, SGI, PGF90...)

The distributed grid class

```
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*.

Scalar field

```
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*. This information can be used to decide when a call to `mpp_update_domains` is required.

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

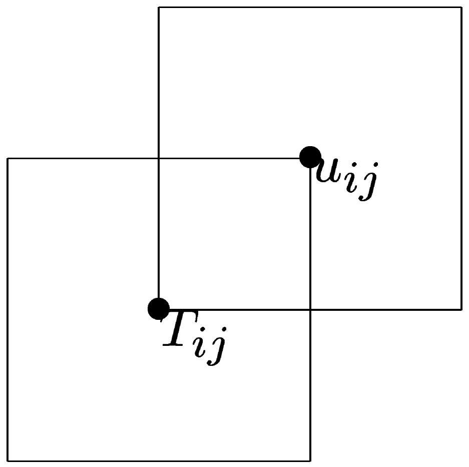
```
  type, public :: hvector2D
     type(scalar2D) :: x, y
     integer :: is, ie, js, je
  end type hvector2D
...
    function add_hvector2D( a, b )
      type(hvector2D) :: add_hvector2D
      type(hvector2D), intent(in) :: a, b
      add_hvector2D%x = a%x + b%x
      add_hvector2D%y = a%y + b%y
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

$$\nabla \cdot \mathbf{u} = \delta_x(\overline{u}^y) + \delta_y(\overline{v}^x) \tag{3}$$

$$(\nabla T)_x = \delta_x(\overline{T}^y) \tag{4}$$

$$(\nabla T)_y = \delta_y(\overline{T}^x) \tag{5}$$



```
    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.”

MEME: Modular Extensible Modeling Environment

- Open class libraries tailored to particular scientific fields offer a way to develop extensible modeling environments for a large multi-institutional user/developer community.
  - Layered approach protects users from unnecessary detail.
  - Classes can be extended without too much pain and suffering.
  - Computational kernels can be added as necessary.
- Requires close collaboration between users, compiler writers, language standards committees.