PARALLEL NUMERICAL KERNELS FOR CLIMATE MODELS

V. BALAJI

SGI/GFDL Princeton University
PO Box 308, Princeton NJ 08542, USA

Climate models solve the initial value problem of integrating forward in time the state of 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. 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.

1 Introduction

Numerical climate and weather models today operate over a wide range of time and space scales. Current resolutions of weather forecasting models approach 50 km for global models, and 10 km for mesoscale models. Ocean models range from $\mathcal{O}(100)$ km for climatic studies to the eddy-resolving models in coastal basins. Non-hydrostatic atmospheric models at kilometre-scale resolutions are used in small domains to study cloud and severe storm dynamics, and in much larger domains spanning $\mathcal{O}(1000)$ km to study the processes underlying large-scale convectively-driven systems such as mid-latitude cyclones and the ITCZ. Depending on the problem under consideration, we may choose to use spectral or grid-point methods; hydrostatic or non-hydrostatic primitive equations; a variety of physical processes may be resolved by the numerics, or remain unresolved (parameterized); the research may be focused on understanding chaotic low-order dynamics of a simplified coupled system, or on a comprehensive accounting of all contributing climate system components; and in all of these, the underlying dynamics remains the solution of the same non-linear Navier-Stokes equation applied to the complex fluids that constitute the planetary climate system.

In climate research, with the increased emphasis on detailed representation of individual physical processes governing the climate, the construction of a model has come to require large teams working in concert, with individual sub-groups each specializing in a different component of the climate system, such as the ocean circulation, the biosphere, land hydrology, radiative transfer and chemistry, and so on. The development of model code now requires teams to be able to contribute components to an overall coupled system, with no single kernel of researchers mastering the whole. This may be called the *distributed development model*, in contrast with the monolithic small-team model of earlier decades.

These developments entail a change in the programming paradigm used in the construction of complex earth systems models. The approach is to build code out of independent modular components, which can be assembled by either choosing a configuration of components suitable to the scientific task at hand, or else easily extended to such a configuration. The code must thus embody the principles of *modularity, flexibility and extensibility.*

The current trend in model development is along these lines, with systematic efforts under way in Europe and the U.S to develop shared infrastructure for earth systems models. It is envisaged that the models developed on this shared infrastructure will go to meet a variety of needs: they will work on different available computer architectures at different levels of complexity, with the same model code using one set of components on a university researcher's desktop, and with a different choice of subsystems, running comprehensive assessments of climate evolution at large supercomputing sites using the best assembly of climate component models available at the moment.

The shared infrastructure currently in development concentrates on the underlying "plumbing" for coupled earth systems models, building the layers necessary for efficient parallel computation and data transfer between model components on independent grids. The next stage will involve building a layer of configurable numerical kernels on top of this layer, and this paper suggests a possible mechanism for the numerical kernel layer.

The Fortran-90 language[1] offers a reasonable compromise between the need to develop high-performance kernels for the numerical algorithms underlying non-linear flow in complex fluids, and the high-level structure needed to harness component models of climate subsystems developed by independent groups of researchers. In this paper, I demonstrate the construction of parallel numerical kernels in F90 in the context of the GFDL Flexible Modeling System (FMS)[a]. The structure of the paper is as follows. Sec. 2is a brief description of FMS. Sec. 3 describes the MPP modules, a modular parallel computing infrastructure underlying FMS, and extensively used in the approach outlined in this paper. In Sec. 4, central to this paper, this shared

[a] http://www.gfdl.gov/~fms

software infrastructure is taken to the next stage: it is shown how configurable numerics can be built, and extended to include numerical algorithms suitable to the problem at hand. A shallow water model is used here as a pedagogical example. The code used in this section is released under the GNU public license (GPL) and is available for download[b]. The section concludes with a discussion of the strengths and limitations of this approach. Sec. 5 summarizes the findings of the paper, and suggests ways forward.

2 FMS: the GFDL Flexible Modeling System

The Geophysical Fluid Dynamics Laboratory (NOAA/GFDL) undertook a technology modernization program beginning in the late 1990s. The principal aim was to prepare an orderly transition from vector to parallel computing. Simultaneously, the opportunity presented itself for a software modernization effort, the result of which is the GFDL Flexible Modeling System (FMS). The FMS constitutes a shared software infrastructure for the construction of climate models and model components for vector and parallel computers. It forms the basis of current and future coupled modeling at GFDL. It has been recently benchmarked on a wide variety of high-end computing systems, and runs in production on three very different architectures: parallel vector (PVP), distributed massively-parallel (MPP) and distributed shared-memory (DSM)[c], as well as on scalar microprocessors. Models in production within FMS include a hydrostatic spectral atmosphere, a hydrostatic grid-point atmosphere, an ocean model (MOM), and land and sea ice models. In development are a non-hydrostatic atmospheric model, an Arakawa C-grid[2] version of the hydrostatic grid-point atmospheric model, an isopycnal coordinate ocean model, and an ocean data assimilation system.

The shared software for FMS includes at the lowest level a parallel framework for handling distribution of work among multiple processors, described in Sec. 3. Upon this are built the *exchange grid* software layer for conservative data exchange between independent model grids, and a layer for parallel I/O. Further layers of software include a *diagnostics manager* for creating runtime diagnostic datasets in a variety of file formats, a *time manager*, general utilities for file-handling and error-handling, and a uniform interface to scientific software libraries providing methods such as FFTs. Interchangeable components are designed to present a uniform interface, so that for instance, behind an ocean "model" interface in FMS may lie a full-fledged ocean model, a few

[b] http://www.gfdl.noaa.gov/~vb/kernels.html
[c] Also known as cache-coherent non-uniform memory access (ccNUMA) architecture.

lines of code representing a mixed layer, or merely a routine that reads in an appropriate dataset, without requiring other component models to be aware which of these has been chosen in a particular model configuration. Physics routines constructed for FMS adhere to the 1D column physics specification[d] providing a uniform physics interface. Coupled climate models in FMS are built as a *single executable* calling subroutines for component models for the atmosphere, ocean and so on. Component models may be on independent logically rectangular (though possibly physically curvilinear) grids, linked by the exchange grid, and making maximal use of the shared software layers.

3 The MPP modules

The parallel framework for modeling in FMS is provided by a layer of software called the MPP modules[e].

Low-level communication. This layer is intended to protect the code from the communication APIs (MPI, SHMEM, shared-memory maps), offering low-overhead protocols to a few communication operations, sufficient for most purposes. It is designed to be extensible to machines which oblige the user to use hybrid parallelism semantics.

The domain class library. This module provides a class[f] to define domain decompositions and updates. It provides methods for performing halo updates on gridpoint models, and data transposes for spectral models or any other purpose (e.g FFTs). It is currently restricted to *logically rectilinear grids* (which category includes non-standard grids such as the bipolar grid[3] and cubed sphere[4]). This layer is described in some detail below.

Parallel I/O. The parallel I/O module provides a simple interface for reading and writing distributed data. It is designed for performance on parallel *writes*, which are far more frequent in these models. Merely by setting the appropriate flags when opening a file, users can choose between different I/O modes, including sequential or direct access, multi-threaded

[d] http://www.gfdl.gov/~fms/gfdl/f90_physics_spec.ps
[e] http://www.gfdl.noaa.gov/~vb/mpp.html
[f] The *class library* terminology is usually associated with object-oriented languages (C++, Java). It is a well-kept secret that F90 modules allow one to build class libraries, having many of the useful features, but few of the current performance disadvantages of OO languages. See POOMA (http://www.mcs.lanl.gov/pooma) for a C++ approach to builing numerical kernels.

or single-threaded I/O[g], writing a single file or multiple files for later assembly. It currently supports netCDF and raw unformatted data, but is designed to be extensible to other formats.

The domain class is briefly described here, as it is used in what follows. The basic element in the hierarchy is an axis specification:

```
type, public :: domain_axis_spec
   integer :: begin, end, size, max_size
   logical :: is_global
end type domain_axis_spec
```

The axis specification merely marks the beginning and end of a contiguous range in index space, plus for convenience some additional redundant information that can be computed from the range. Using this, we construct a derived type called `domain1D` that can be used to specify a 1D domain decomposition:

```
type, public :: domain1D
   type(domain_axis_spec) :: compute, data, global
   integer :: pe
   type(domain1D), pointer :: prev, next
end type domain1D
```

The domain decomposition consists of three axis specifications that contain all the information about the grid topology necessary for communication operations on distributed arrays. The *compute domain* specifies the range of indices that will be computed on a particular processing element (PE). The *data domain* specifies the range of indices that are necessary for these computations, i.e by including a halo region sufficiently wide to support the numerics. The *global domain* specifies the global extent of the array that has been distributed across processors. In addition the `domain1D` type associates a processor ID with each domain, and maintains a linked list of neighbours in each direction.

It is now simple to construct higher-order domain decompositions, of which the 2D decomposition is the most common. This is specified by a derived type called `domain2D`, which is constructed using orthogonal `domain1D` objects:

```
type, public :: domain2D
   type(domain1D) :: x, y
   integer :: pe
   type(domain2D), pointer :: west, east, south, north
```

[g] Single-threaded I/O on multiple processors involves having one processor gather the global data for writing, or scatter the data after reading.

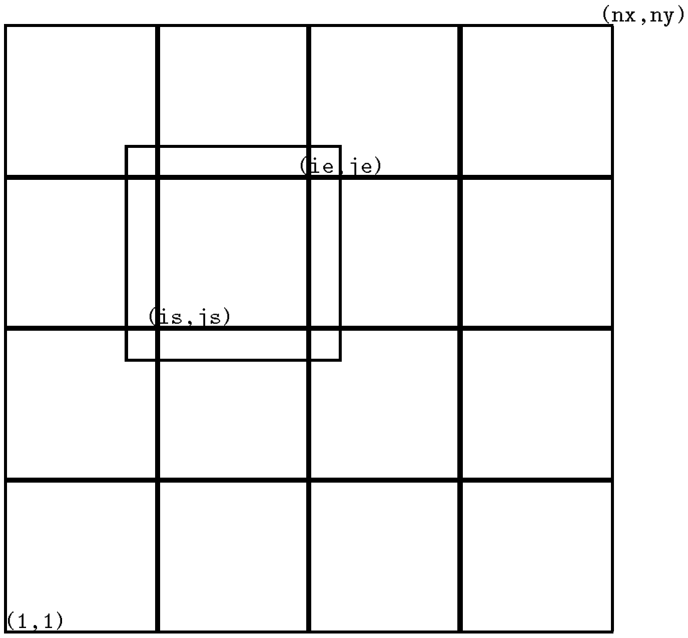

Figure 1. The `domain2D` type, showing global, data and compute domains.

```
end type domain2D
```

Many of the methods associated with a `domain2D` can be assembled from operations on its `domain1D` elements. The additional information here is to maintain a list of neighbours along two axes, and to assign a PE to each 2D domain. The `domain2D` object is shown here in Fig. 1, for a global domain `(1:nx,1:ny)` distributed across processors and yielding a compute domain `(is:ie,js:je)`.

There are two basic operations on the domain class. One is to set up a domain decomposition based on the model grid topology. The second is a high-level communication operation to fill in non-local data points which lie in compute domains on other PEs, and must therefore be updated after a compute cycle (e.g halo update).

The simplest version of the first call is given a global grid and the required halo sizes in x and y, and returns an array of `domain2D` objects with the required decomposition:

```
type(domain2D) :: domain(0:npes-1)
call mpp_define_domains( (/1,nx,1,ny/), domain, xhalo=2, yhalo=2 )
```

This requests a decomposition of the global domain on `npes` PEs, with a halo size of 2. The layout across PEs and domain extents here are internally chosen, though optional arguments allow the user to take control of these if desired. In addition, there are optional flags for the user to pass in information about the grid topology, e.g periodic boundaries in x or y, or folds (as in cylindrical or bipolar grids).

After setting up a domain decomposition, we may proceed with a computation. All data is allocated on the *data* domain, of which only the compute domain subset is locally updated. At the end of a compute cycle, typically a timestep, we make a call to update the non-local data (halo region):

```
real, allocatable :: f(:,:)
call mpp_define_domains( (/1,nx,1,ny/), domain, xhalo=2, yhalo=2 )
call mpp_get_compute_domain( domain, is, ie, js, je )
call mpp_get_data_domain( domain, isd, ied, jsd, jed )
allocate( f(isd:ied,jsd:jed) )
do t = start,end              !time loop
   do j = js,je
      do i = is,ie
         f(i,j) = ...         !computational kernel
      end do
   end do
   call mpp_update_domains( f, domain )
end do
```

All the topology and connectivity information is stored in `domain`, allowing the halo update to proceed. This data encapsulation proves to be a powerful mechanism to simplify high-level code. The syntax remains identical for arrays of any type, kind or rank; for open or periodic boundary conditions; for simple or exotic logically rectangular grids; and on a variety of parallel computing architectures. It is a testament to the success of the layered and encapsulated interface design that there are very few circumstances under which developers of parallel algorithms in FMS have needed direct recourse to the underlying communication protocols.

In the next section we demonstrate the construction of a layer of numerical kernels on top of the `domain` class library.

4 Parallel numerical kernels in F90

The shallow water model represents the dynamics of small displacements of the surface of a fluid layer of constant density in a gravitational field. Given a rotating fluid of height H, the 2D dynamics of a small displacement $\eta \ll H$

may be written as:

$$\frac{\partial \eta}{\partial t} = -H\nabla \cdot \mathbf{u} + \nu\nabla^2\eta \tag{1a}$$

$$\frac{\partial \mathbf{u}}{\partial t} = -g\nabla\eta + f\mathbf{k}\times\mathbf{u} + \nu\nabla^2\mathbf{u} + \mathbf{F} \tag{1b}$$

where $\mathbf{u}$ is the horizontal velocity, g the acceleration due to gravity, f the Coriolis parameter, ν the coefficient of viscosity, and $\mathbf{F}$ an external forcing. I have chosen to discretize it below using a forward-backward timestep, and an explicit treatment of rotation, as follows:

$$\frac{\eta^{n+1}-\eta^n}{\Delta t} = -H(\nabla\cdot\mathbf{u})^n + \nu\nabla^2\eta^n \tag{2a}$$

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

The superscript refers to the time level. The spatial discretization has been left unspecified because the numerical kernels constructed will be able to support multiple grid stencils. Observant readers will note that the treatment of the Coriolis term is formally unstable; the unconditionally stable representation introduces some additional considerations that are discussed below in Sec. 4g.

The shallow water model is a standard pedagogical tool used in NWP[5]. It is also of significance in real models: each layer of a mass-based vertical coordinate system may be thought of as being a shallow water layer. In addition, it is customary in ocean models to separate out the 2D barotropic dynamics of the ocean, which is formally a shallow water model, from the 3D baroclinic dynamics of the interior, whose intrinsic timescales are much longer. Parallel barotropic solvers tend to be latency-bound and limit the scalability of ocean codes[6], and are thus a potential client for efficient numerical kernels developed for a shallow water model.

4a A class hierarchy for the shallow water model

The key to constructing class libraries that are intuitive and easy to use is to define objects as close as is reasonable to the elements of the system being modeled. Here the objects being modeled are vector and scalar fields, and the operations are arithmetic, rotational and differential operations on fields. These will thus become the objects and methods of our class. Differential operations on scalar and vector fields will require a representation of a *grid*, with an associated metric tensor. Finally, for parallel codes, the grid will need to built on a layer supporting domain decomposition. Thus the class hierarchy being developed is:

The domain class described above, representing index-space topologies of domains distributed across parallel processing elements;

A grid class superposing a physical grid on the domain class, supplying a grid metric tensor and differentiation coefficients;

A field class of 2D scalar and vector fields supporting arithmetic, rotational, and differential operations on the grid.

The shallow-water model is 2D, and therefore tends to be latency-bound, since a 2D halo region produces a short communication byte-stream. We use the *active domain* axis specification for latency-hiding.

4b Active domains.

A new axis specification that has recently been added to the domain class is the *active domain* axis specification. This can be used to maintain the state of the data domain, and can provide a simple and powerful mechanism of trading communication for computation on parallel clusters with high-latency interconnect fabrics, or, more generally, on high-latency code segments. The idea is to use *wide halos* and compute as much locally in the halo as the numerics permit. These points are shared by more than one PE, and thus are redundantly computed. The redundant computations permit one to perform halo updates less often. In each computational cycle, the size of the active domain is reduced until no further computations are possible without a halo update. Only then is the halo update performed. An example is shown below in Sec. 4f. Fig. 2 shows successive stages of a computational cycle requiring halo updates on every fourth timestep.

4c Scalar and vector fields

The scalar field is constructed as follows:

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

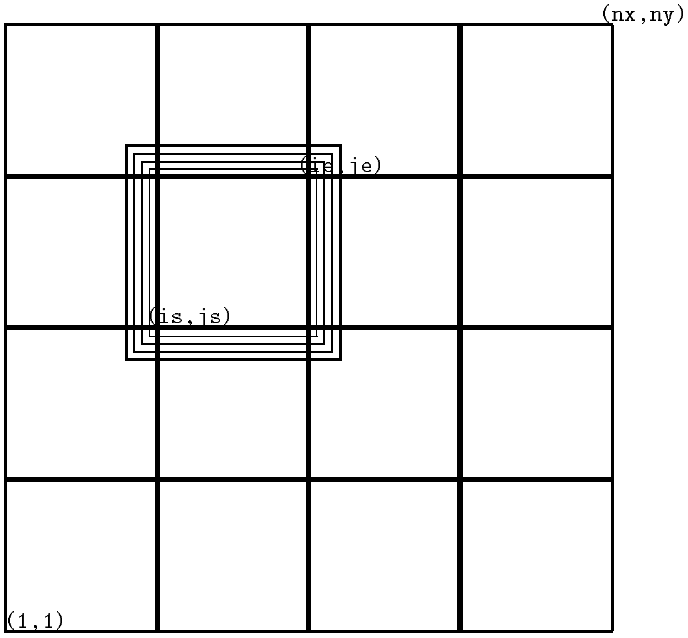

Figure 2. The `domain2D` type, showing global, data and compute domains as thick lines. Successive stages of a diminishing active domain are shown as thin lines.

The `data` element of the `scalar2D` type contains the field values. The Fortran standard requires arrays within derived types to have the `pointer` attribute. (This will be revised in Fortran-2000, which will permit allocatable arrays within types). The active domain described in Sec. 4b maintains the state of the data domain, and sets the limits of the domain that contain valid data. We use this information to limit the array sections used for computations, and to determine when a halo update is required.

There are pros and cons to the use of arrays with a `pointer` attribute:

- Pointer arrays can be pointed to a persistent private workspace, called the *user stack*. This has two advantages:
  - It saves the overhead of a system call to allocate memory from the heap;
  - It avoids the possibility of memory leaks that can be caused by incautious use of the `allocate` statement on `pointer` arrays:

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

At this point, the original allocation of 100 words has no handle, and cannot be deallocated by the user, nor can the compiler determine whether or not some array is pointing to this location. This is a memory leak that can grow without limit, if it occurs in a routine that is repeatedly called.

- It may not be possible for an optimizing compiler to determine if two pointer arrays are or are not aliased[h] to each other. Non-standard compiler directives (e.g `!dir$ IVDEP` on compilers supporting Cray directive syntax) generally permit the user to assist the compiler in this regard.
- The F90 standard requires the assignment operator (=) applied to pointer components to redirect them, rather than copy them. This can yield counter-intuitive behaviour:

```
type(scalar2D) :: a, b
a = b
b%data = ...
```

This will change the values of `a%data`, which may run against expectations. (Changes to the RHS of an assignment *after* the assignment do not affect the LHS for ordinary variables). A further complication is that the future revision of the standard, which permits allocatable array components, will restore the expected behaviour.

In consideration of the last issue, we overload the assignment interface to restore the expected behaviour when the RHS is of `type(scalar2D)`. We also use the assignment interface to make it possible to assign simple arrays or scalars to a scalar field (e.g `a=0`):

```
interface assignment(=)
   module procedure copy_scalar2D_to_scalar2D
   module procedure assign_0D_to_scalar2D
   module procedure assign_2D_to_scalar2D
end interface
```

[h] i.e, have overlapping memory locations.

In all versions of these procedures, we first allocate space for the LHS data from the user stack. In the first instance of the overloaded assignment, the RHS is also a scalar field, and the values in its `data` element are copied into the LHS `data`, and the active domain on the RHS is applied on the result. In the other two instances, the RHS is a real array or scalar, which is used to fill the `data` element of the LHS:

```
type(scalar2D) :: a, b
real :: c
real :: d(:,:)
a = b
a = c
a = d
```

The different possibilities are illustrated above. In the assignment of `b` and `d`, an error results if the arrays on the RHS do not conform to `a%data`.

The horizontal vector field is constructed as a pair of scalar fields:

```
type, public :: hvector2D
   type(scalar2D) :: x, y
   integer :: is, ie, js, je
end type hvector2D
```

and its assignment operations are inherited from its scalar components:

```
subroutine copy_hvector2D_to_hvector2D( a, b )
  type(hvector2D), intent(inout) :: a
  type(hvector2D), intent(in)    :: b
  a%x = b%x
  a%y = b%y
  return
end subroutine copy_hvector2D_to_hvector2D
```

4d Arithmetic operators

Consider the addition operation applied to scalar fields:

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

This operation requires the arithmetic operation `a%data = b%data + c%data`, subject to limits on the active domain. The active domain that results is the intersection of the two active domains on the LHS:

```
interface operator(+)
   module procedure add_scalar2D
   module procedure add_hvector2D
```

```
end interface

function add_scalar2D( a, b )
  type(scalar2D) :: add_scalar2D
  type(scalar2D), intent(in) :: a, b
  add_scalar2D%data => work2D(:,:,nbuf2) !assign from user stack
  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)
  do j = add_scalar2D%js,add_scalar2D%je
     do i = add_scalar2D%is,add_scalar2D%ie
        add_scalar2D%data(i,j) = a%data(i,j) + b%data(i,j)
     end do
  end do
  return
end function add_scalar2D
```

For multiplication, one of the multiplicands is a scalar or 2D array.

The vector field is able to inherit its arithmetic operations as a combination of operations on its components, just as was done for the assignment interface in Sec. 4c.

4e Rotational operators

The computation of the Coriolis term requires the rotation of a vector field about the vertical. We can define an operator `kcross` which takes a single vector field argument and returns a rotated vector field. This is very striaghtforward on grid stencils where the vector components are defined at the same point, as in the Arakawa A and B grids[2], but requires averaging for the C-grid, where vector components are not collocated.

The function `kdotcurl`, taking a vector field operand and returning the `k`-component of vorticity as a scalar field, may additionally be defined if needed.

4f Differential operators

The differential operators include *gradient*, which takes a scalar field operand and returns a vector field; *divergence*, taking a vector field operand and returning a scalar field, and *laplacian*, defined for both vector and scalar fields. I illustrate here the construction of differential operators on a B-grid. The B-grid T- and U-cells are shown in Fig. 3:

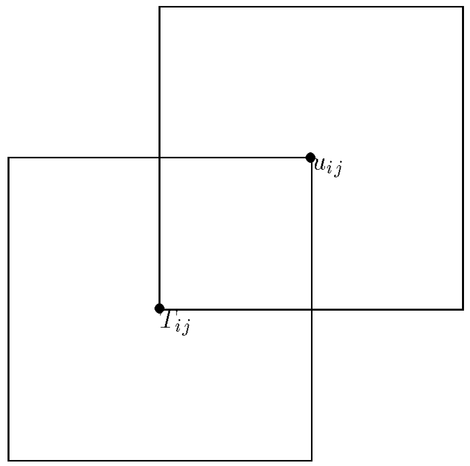

Figure 3. T- and U-cells on the Arakawa B-grid.

The discrete representation of gradient and divergence operators using forward differences on the B-grid is:

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

$$(\nabla T)_x = \delta_x(\overline{T}^y) \tag{3b}$$

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

The code for the gradient operator is shown below:

```
function grad(scalar)
  type(hvector2D) :: grad
  type(scalar2D), intent(inout) :: scalar

  call mpp_get_compute_domain( domain, is, ie, js, je )
  call mpp_get_data_domain( domain, isd, ied, jsd, jed )
  grad%x => work2D(:,:,nbufx)
  grad%y => work2D(:,:,nbufy)
  if( scalar%ie.LE.ie .OR. scalar%je.LE.je )then
      call mpp_update_domains( scalar%data, domain, EAST+NORTH )
      scalar%ie = ied
      scalar%je = jed
  end if
  grad%is = scalar%is; grad%ie = scalar%ie - 1
  grad%js = scalar%js; grad%je = scalar%je - 1
  do j = grad%js,grad%je
     do i = grad%is,grad%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,nbufx) = gradx(i,j)*( tmp1 + tmp2 )
      work2D(i,j,nbufy) = grady(i,j)*( tmp1 - tmp2 )
   end do
end do
```

There are certain aspects of this code fragment worth highlighting:

- The differencing on the B-grid requires the values at `(i+1,j+1)` in order to compute `(i,j)`, thus losing one point in the eastern and northern halos on each compute cycle. A halo update is performed when there are insufficient active points available to fill the compute domain of the resulting vector field. We may avoid updating the western and southern halos, which are never used in this computation.

  The halo size must be at least 1 for this numerical kernel, but can be set much higher at initialization in the `mpp_define_domains` call (Sec. 3). For a halo size of `n`, halo updates are required only once every `n` timesteps. This illustrates the sorts of parallel optimizations yielded by the construction of modular numerical kernels.

- The loop itself has a complicated form, particular to the B-grid stencil shown. Other loop optimizations may be applied here without compromising the higher-level code. Operators for other stencils may be overloaded here as well under the generic interface `grad`, again preserving higher-level code structure.

- The halo updates have been illustrated here as *blocking* calls, which complete upon return from the call. For a fully non-blocking halo update, the call would be made at the end of a compute cycle, and the usual wait-for-completion call (e.g `MPI_WAIT`) called at the top of the next cycle.

4g High-level formulation of a shallow water model

Using the constructs developed here, a basic shallow water model may be written in standard Fortran-90 as follows:

```
program shallow_water
use fields
type(scalar2D) :: eta
type(hvector2D) :: u, forcing
do t = start,end           !time loop
 eta = eta + dt*( -h*div(u) + nu*lapl(eta) )
```

```
 u = u + dt*( -g*grad(eta) + f*kcross(u) + nu*lapl(u) + forcing )
end do
end program shallow_water
```

This code[i] possesses several of the features we seek to build into our codes: portability, scalability, ease of use, modularity, flexibility and extensibility:

- The code has been validated on Cray PVP and MPP systems, SGI cc-NUMA, and a Beowulf cluster using the PGF90 compiler. On the Cray and SGI compilers, the abstraction penalty without using special directives is estimated at about 20%. Some causes for the abstraction penalty are noted below.
- For a typical model grid of 200×200 points, it scales to 80% on 64 PEs with a halo size of 1. It is memory-scaling except for the halo region.
- The parallel calls are built into the kernels, and are called only at need. Both blocking and non-blocking halo updates are supported. In addition, the frequency of halo updates can be reduced on a high-latency interconnect at the cost of additional computation, merely by setting wider halos at initialization, and no other code changes.
- The grid stencils, and the ordering of indices, etc., can be changed within the kernels without affecting the calling routines. This is done by overloading kernels for different grid stencils under the same generic interface. In addition this would permit the use of different grid stencils in different parts of the code. However, there are some basic limits to this flexibility, noted below.

Some difficulties with this approach may also be noted. One is that arithmetic operators constructed as shown here do not perform certain optimizations that would be done for equivalent array operations, illustrated by these examples:

```
a = a + b
a = b + c + d
```

In the first example, one memory stream may be saved by knowing that the LHS array also appears on the RHS. Since a function call is used to perform the addition here, an extra memory stream is used. In the second example, the loop construct would chain the two additions together in a single loop. Here the functions are called pairwise. While it is possible in principle to

[i] Available for download from http://www.gfdl.noaa.gov/~vb/kernels.html

chain function calls, it is unlikely that any compiler in practice performs this level of interprocedural analysis. These are contributors to the abstraction penalty noted above.

A second limitation of this approach affects the flexibility in the choice of grid stencil for a given high-level casting of the equations. This is due to the fact that the time discretization must be explicitly maintained in high-level code while we seek to construct numerical kernels for spatial operators. This is best illustrated by modifying Eq. 2 to use an implicit treatment of the Coriolis force:

$$\frac{\eta^{n+1} - \eta^n}{\Delta t} = -H(\nabla \cdot \mathbf{u})^n \tag{4a}$$

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} = -g(\nabla \eta)^{n+1} + f\mathbf{k} \times \left(\frac{\mathbf{u}^{n+1} + \mathbf{u}^n}{2} \right) + \mathbf{F}^n \tag{4b}$$

Using a grid stencil where vector components are collocated, it is easy to rearrange terms in the $\mathbf{u}$ equation to bring $\mathbf{u}^{n+1}$ to the LHS. The timestepping for $\mathbf{u}$ then takes place in two steps:

```
f2 = f/2
u = u + dt*( -g*grad(eta) + f2*kcross(u) + nu*lapl(u) + forcing )
u = ( u + dt*f2*kcross(u) )/( 1 + f2**2 )
```

However, on a C-grid, since the vector components are not collocated, the implicit Coriolis term requires a matrix inversion[7]. Particularly vexing in the current context is that the high-level structure of the equations depends on the spatial grid stencil. The high-level form of the equations may not in all instances be able to be made independent of the grid numerics.

5 Discussion

Earth systems models are run over a wide range of time and space scales, in many configurations, serving a wide variety of needs, while sharing the same underlying Navier-Stokes dynamics applied to complex fluids. As model complexity grows, the need for a flexible, modular and extensible high-level structure to support a distributed software development model has become apparent.

First attempts at standardization for parallel computing architecture resulted in standards for low-level communication APIs (MPI, OpenMP). These have the disadvantage of implying a particular kind of underlying hardware (distributed or shared memory) and add greatly to code complexity for portable software that attempts to encompass multiple parallelism semantics.

Efforts are currently underway to create a software infrastructure for the climate modeling community that meets these needs with high-performance code running on a wide variety of parallel computing architectures, concealing the communication APIs from high-level code. A prototype for this would be the domain class library described in Sec. 3.

The class library approach is a powerful means of encapsulating model structure in an intuitive manner, so that the data structures being used are objects conceptually close to the entities being modeled. In this paper I have demonstrated the extension of this approach to create a class of modular, configurable parallel numerical kernels using Fortran-90 suitable for use in earth systems models. The language offers a reasonable compromise between high-performance numerical kernels and the high-level structure required for a distributed development model. This approach is being experimentally applied to production models, particularly in high-latency code segments where configurable halos could yield performance enhancements. It could yield additional advantages in models where we may need to choose different grid stencils for different research problems using the same model code.

The parallel numerical kernels developed here have been validated on a small subset of current compilers and architectures. As they use many advanced language features, it is likely that performance on different platforms will be uneven. This is something that the developers of shared software infrastructure will have to come to terms with. A key recommendation flowing from the conclusions of this paper would be an earnest effort on our part to involve the compiler and language standards communities in these efforts. This involves closer partnership, rather than an antagonistic relationship, with the scalable computing industry, and participation of our community in the evolution of language standards.

Acknowledgments

This work is a part of the research and development undertaken by the FMS Development Team at NOAA/GFDL to build a software infrastructure for modeling.

Bill Long of Cray Inc. provided valuable assistance and advice in elucidating issues of F90 standard compliance, implementation and optimization in an actual compiler (CF90).

Thanks to Jeff Anderson, Ron Pacanowski and Bob Hallberg of GFDL for close examination and critique of this work at each stage of development, including the final manuscript.

References

1. Michael Metcalf and John Reid. *Fortran 90/95 Explained.* Oxford University Press, 2nd edition, 1999.
2. A. Arakawa. Computational design for long-term numerical integration of the equations of atmospheric motion. *J. Comp. Phys.*, 1:119–143, 1966.
3. Ross J. Murray. Explicit generation of orthogonal grids for ocean models. *J. Comp. Phys.*, 126:251 – 273, 1996.
4. M. Rancic, R.J. Purser, and F. Mesinger. A global shallow-water model using an expanded spherical cube: Gnomonic versus conformal coordinates. *Quart. J. Roy. Meteor. Soc.*, 122:959 – 982, 1996.
5. G. J. Haltiner and R. T. Williams. *Numerical Prediction and Dynamic Meteorology.* John Wiley and Sons, New York, 1980.
6. Stephen M. Griffies, Ronald C. Pacanowski, Martin Schmidt, and V. Balaji. Tracer conservation with an explicit free surface method for z-coordinate ocean models. *MWR*, 129:1081–1098, 2001.
7. D.E. Dietrich, M.G. Marietta, and P.J. Roache. An ocean modeling system with turbulent boundary layers and topography, part 1: numerical description. *Int. J. Numer. Methods Fluids*, 7:833–855, 1987.