**The Flexible Modeling System**

V. Balaji, GFDL/Princeton University

Zhi Liang, GFDL/HPTI

1. **Introduction: the emergency of modeling framework.**

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 development process of earlier decades.

A simultaneous trend is the increase in hardware and software complexity in high-performance computing, as we shift toward the use of scalable computing architectures. Scalable architectures come in several varieties, including shared-memory parallel vector systems, distributed memory massively-parallel systems. The individual computing elements themselves can embody complex memory hierarchies, especially in the recent trend toward multi-core and many-core systems, co-processors and accelerators. To facilitate sharing of code and development costs across multiple institutions, it is necessary to abstract away the details of the underlying architecture and provide a uniform programming model across different scalable and uniprocessor architectures.

|  |
| --- |
| Coupler Layer  FMS Superstructure |
| Model Layer  User codes (component models ) |
| Distributed grid Layer  FMS Infrastructure |
| Machine Layer |

Figure 1.1: The architecture of Flexible Modeling System

The GFDL Flexible Modeling System (FMS) is an early example of a modeling framework, a comprehensive programming model and toolkit for the construction of coupled climate models. The schematic description of the FMS architecture is shown in Fig. 1.1. This “sandwich” architecture is fairly typical of such frameworks. User code, that is to say a set of routines expressing scientific algorithms, is written following the conventions of a standard infrastructure layer that provides useful and common technical services such as I/O, exception handling, and most importantly, operations on distributed grids and fields. Such standard high level expressions of parallelism, independent of the underlying hardware architecture, and uniformly expressed on all platforms, are an area of keen research interest.

This chapter is organized as follows. In Section 2 we describe the FMS superstructure. In Section3, we will summarize the basic components of FSM infrastructure.

1. **FMS superstructure**

**2.1 Coupler**

The FMS coupler is designed to address the question of how different components of the

Earth system, say atmosphere and ocean, are discretized. Earlier generations of climate models used the same discretization, or simple integer refinement, for all components: thus, data exchange between components was a relatively simple point-to-point exchange. But any limitation on resolution of one component necessarily imposed itself on the other as well. Now it is increasingly common for each model component to make independent discretization choices appropriate to the particular physical component being modeled. In this case, how is, say a sea surface temperature from an ocean model made available to an atmosphere model that will use it as a boundary condition on a different spatial grid?

This is the regridding problem, subject to the following constraints when specialized t

earth system models:

* Quantities must capable of being globally conserved: if there is a flux of a quantity across an interface, it must be passed conservatively from one component to the other. This consideration is less stringent when modeling weather or short-term (intraseasonal to interannual) climate variability, but very important in models of secular climate change, where integration times can be O(1000,000) − O(100,000,000) timesteps.
* The numerics of the flux exchange must be stable, so that no limitation on the individual component time step is imposed by the boundary flux computation itself.
* There must be no restrictions on the discretization of a component model. In particular, resolution or alignment of coordinate lines cannot be externally imposed. This also implies a requirement for higher-order interpolation schemes, as low-order schemes work poorly between grids with a highly skewed resolution ratio. Higher-order schemes may require that not only fluxes, but their higher-order spatial derivatives as well, be made available to regridding algorithms.
* The independent discretization requirement extends to the time axis: component models may have independent time steps. (We do have a current restriction that a coupling time step be an integral multiple of any individual model time step, and thus, time steps of exchanging components may not be co-prime).
* The exchange must take place in a manner consistent with all physical processes occurring near the component surface. This requirement is highlighted because of the unique physical processes invoked near the planetary surface: in the atmospheric and oceanic boundary layers, as well as in sea ice and the land surface, both biosphere and hydrology.
* Finally, we require computational efficiency on parallel hardware: a solution that is not rate-limiting at the scalability limits of individual model components. Components may be scheduled serially or concurrently between coupling events.

The FMS coupler recognizes only a few components that may be on independent grids:

an atmosphere, an ocean surface, a land surface, and an ocean. The ocean surface also represents the sea ice. Any other components inherit a grid from these, e.g atmospheric physics and chemistry from the atmosphere; terrestrial biosphere, river and land ice components from the land surface; marine biogeochemistry from the ocean.

There are interfaces or “slots” for each of the specific components listed above. For instance, an ocean model would encode its state in terms of specific data structures to hold the fields it exchanges with other components, called ocean boundary type and ocean data type. It must provide calls named ocean model init and ocean model end for initialization

and termination, and a routine called update ocean model that steps the model forward for one coupling timestep. These calls all have a specific syntax. Each slot also includes the possibility of a null or “stub” component if that component is not needed, as well as a “data” component (where for instance the ocean is replaced by a dataset). In addition we provide a data override capability for fine-tuned sensitivity studies, where individual fields in the model can be overridden by a dataset.

Code Block 1.1 shows an example of such a data structure, used to exchange data between ice and ocean components. The type is composed principally of a number of 2D surface fields, and the variable xtype which encodes the type of exchange.

type, public :: ice ocean boundary type

real :: u flux(:,:) ! wind stress (Pa)

real :: v flux(:,:) ! wind stress (Pa)

real :: t flux(:,:) ! sensible heat flux (W/m2)

real :: q flux(:,:) ! specific humidity flux (kg/m2/s)

real :: salt flux(:,:) ! salt flux (kg/m2/s)

real :: lw flux(:,:) ! long wave radiation (W/m2)

real :: sw flux vis dir(:,:) ! direct visible sw radiation (W/m2)

real :: sw flux vis dif(:,:) ! diffuse visible sw radiation (W/m2)

real :: sw flux nir dir(:,:) ! direct near IR sw radiation (W/m2)

real :: sw flux nir dif(:,:) ! diffuse near IR sw radiation (W/m2)

real :: lprec(:,:) ! mass flux of liquid precip (kg/m2/s)

real :: fprec(:,:) ! mass flux of frozen precip (kg/m2/s)

real :: runoff(:,:) ! mass flux of liquid runoff (kg/m2/s)

real :: calving(:,:) ! mass flux of frozen runoff (kg/m2/s)

real :: runoff hflx(:,:) ! heat flux of liquid land water (W/m2)

real :: calving hflx(:,:) ! heat flux of frozen land water (W/m2)

real :: p(:,:) ! pressure of sea ice and atmosphere (Pa)

integer :: xtype ! REGRID, REDIST or DIRECT

type(coupler 2d bc type) :: fluxes ! additional tracers

end type ice ocean boundary type

Code Block 1.1: ice ocean boundary type

Coupler is the driver that couples component models for atmosphere, ocean, land, and sea ice on independent grids. It encapsulates boundary state and boundary fluxes. The component models are coupled to allow implicit vertical diffusion of heat and moisture at the interfaces of the atmosphere, land, and ice models. As a result, the atmosphere, land, and ice models all use the same time step. The atmospheric model has been separated into down and up calls that correspond to the down and up sweeps of the standard tri-diagonal elimination. The ocean interface uses explicit mixing. Fluxes to and from the ocean must be passed through the ice model. This includes atmospheric fluxes as well as fluxes from the land to the ocean (runoff). This program contains the model's main time loop. Each iteration of the main time loop is one coupled (slow) time step. Within this slow time step loop is a fast time step loop, using the atmospheric time step. Exchange between sea ice and ocean occur once every slow time step. Code block 1.2 is the slow time step. Fast loop is executed inside the slow time step and is listed in Code block 1.3.

do nc = 1, num\_cpld\_calls

call generate\_sfc\_xgrid( Land, Ice )

call flux\_ocean\_to\_ice( Ocean, Ice, Ocean\_ice\_flux )

call update\_ice\_model\_slow\_up( Ocean\_ice\_flux, Ice )

!fast loop

call update\_land\_model\_slow(Land)

call flux\_land\_to\_ice( Land, Ice, Land\_ice\_flux )

call update\_ice\_model\_slow\_dn( Atmos\_ice\_flux, Land\_ice\_flux, Ice )

call flux\_ice\_to\_ocean( Ice, Ice\_ocean\_flux )

call update\_ocean\_model( Ice\_ocean\_flux, Ocean )

enddo

Code block 1.2: source code of slow time step

do na = 1, num\_atmos\_calls

Time = Time + Time\_step\_atmos

call sfc\_boundary\_layer( Atm, Land, Ice, &

Land\_ice\_atmos\_flux )

call update\_atmos\_model\_down( Land\_ice\_atmos\_flux, Atm )

call flux\_down\_from\_atmos( Time, Atm, Land, Ice, &

Land\_ice\_atmos\_flux, Atmos\_land\_flux, Atmos\_ice\_flux )

call update\_land\_model\_fast( Atmos\_land\_flux, Land )

call update\_ice\_model\_fast( Atmos\_ice\_flux, Ice )

call flux\_up\_to\_atmos( Time, Land, Ice, Land\_ice\_atmos\_flux )

call update\_atmos\_model\_up( Land\_ice\_atmos\_flux, Atm )

enddo

Code block 1.3: source code of fast time step

The coupler also supports both serial coupling and concurrent coupling. Serial coupling uses a forward-backward time step for coupling, which is illustrated in figure (1.2)



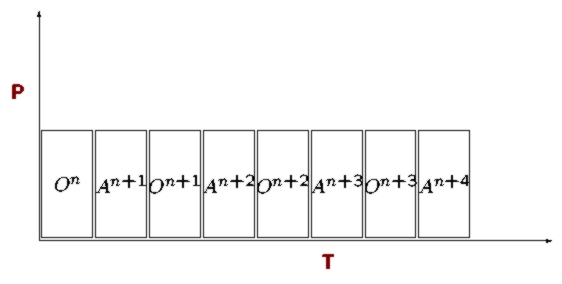


Figure 1.2: Serial coupling

The concurrent coupling uses a forward-only time step for coupling, which is illustrated in figure 1.3. While this is unconditionally unstable, the system is strongly damped. Answers vary with respect to serial coupling, as the ocean is now forced by atmospheric state from one step ago. The advantage of concurrent coupling is improving scalability and performance.



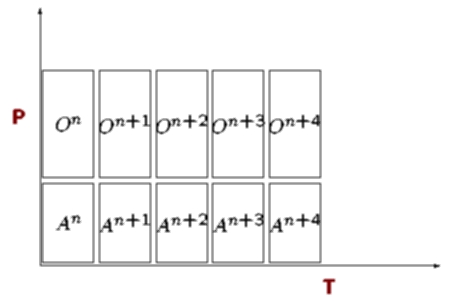
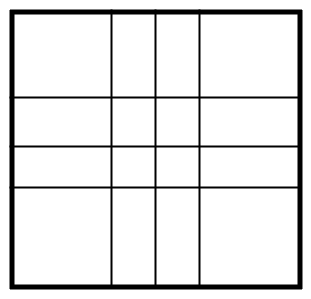
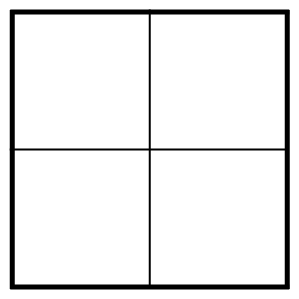
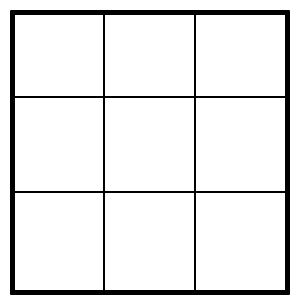


Figure 1.3: concurrent coupling

**2.2 Exchange grid.**

Exchange grid is defined as a set of cells created by edges joining pairs of vertices defined in a discretization. Given two grids, an exchange grid is the set of cells defined by the union of all the vertices of the two parent grids. This is illustrated in Fig. 1.4, with two parent grids (“atmosphere” and “ocean”). As seen here, each exchange grid cell can be uniquely associated with exactly one cell on each parent grid, and fractional areas with respect to the parent grid cells. Quantities being transferred from one parent grid to the other are first interpolated onto the exchange grid using one set of fractional areas; and then averaged onto the receiving grid using the other set of fractional areas. If a particular moment of the exchanged quantity is required to be conserved, consistent moment-conserving interpolation and averaging functions of the fractional area may be employed. This may require not only the cell-average of the quantity (zeroth-order moment) but also higher-order moments to be transferred across the exchange grid.



Atmosphere

Ocean

Exchange

Figure 1.4: Exchange grid between atmosphere and ocean grid

Given N cells of one parent grid, and M cells of the other, the exchange grid is, in the limiting case in which every cell on one grid overlaps with every cell on the other, a matrix of size N × M. In practice, however, very few cells overlap, and the exchange grid matrix is extremely sparse. In code, we typically treat the exchange grid cell array as a compact 1D array with indices pointing back to the parent cells. Table 1.1 shows the characteristics of exchange grids at typical climate model resolutions. The first is the current GFDL model CM2.1, and the second for CM2.4, a higher-resolution version. As seen here, the exchange grids are extremely sparse.

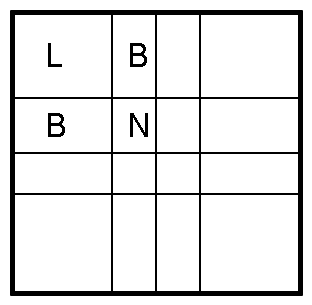
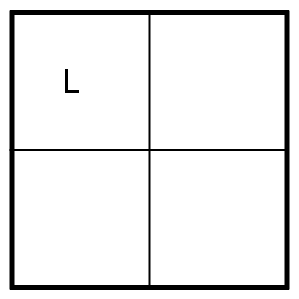
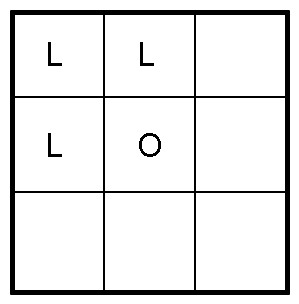
|  |  |  |
| --- | --- | --- |
| Atmosphere | Ocean | Xgrid |
| 144x90 | 360x200 | 79644 |
| 288x180 | 1080x840 | 895390 |

Table 1.1: Exchange grid sizes for typical climate model grids.

The computation of the exchange grid itself could be time consuming, for parent grids on completely non-conformant curvilinear coordinates. In practice, this issue is often sidestepped by pre-computing and storing the exchange grid. The issue must be revisited if either of the parent grids is adaptive.

The FMS implementation of exchange grids restricts itself to 2-dimensional grids on the planetary surface. However, there is nothing in the exchange grid concept that prevents its use in exchanges between grids varying in 3, or even 4 (including time) dimensions.

A complication arises when one of the surfaces is partitioned into complementary components: in Earth system models, a typical example is that of an ocean and land surface that together tile the area under the atmosphere. Conservative exchange between three components may then be required: crucial quantities like CO2 have reservoirs in all three media, with the total carbon inventory being conserved.



Land

Ocean

Exchange

Fig. 1.5: The mask problem. The land and atmosphere share the grid on the left, and their discretization of the land-sea mask is different from the ocean model, in the right. The exchange grid, middle, is where these may be reconciled: the “orphan” cell marked N is assigned (arbitrarily) to the land, and the land cell areas “clipped” to remove the doubly-owned cells marked B.

Figure 1.5 shows such an instance, with an atmosphere-land grid and an ocean grid of different resolution. The darker line in the first two frames shows the land-sea mask as discretized on the two grids, with the cells marked L belonging to the land. Due to the differing resolution, certain exchange grid cells have ambiguous status: the two cells marked B are claimed by both land and ocean, while the orphan cell marked N is claimed by neither, as it is considered land by the ocean component and ocean by the land component.

This implies that the mask defining the boundary between complementary grids can only be accurately defined on the exchange grid: only there can it be guaranteed that the cell areas exactly tile the global domain. Cells of ambiguous status are resolved here, by adopting some ownership convention. For example, in the FMS exchange grid, we generally modify the land model as needed: the land grid cells are quite independent of each other and amenable to such transformations. We add cells to the land grid until there are no orphan cells left on the exchange grid, then get rid of the doubly-owned cells by clipping the fractional areas on the land side.

**3 FMS Infrastructure**

FMS is a software infrastructure for constructing and running atmospheric, oceanic, and climate system models. This infrastructure includes software to handle parallelization, input and output, data exchange between various model grids. This infrastructure should largely insulate FMS users from machine-specific details. This infrastructure includes software to handle communication (section 3.1), domain decomposition (section 3.2), parallel input/output (section 3.3), diagnostics(section 3.4) and data\_override(section 3.5).

3.1 Mpp\_mod

mpp\_mod, is a set of simple calls to provide a uniform interface to different message-passing libraries. It currently can be implemented in the MPI standard.

The data transfer between a processor and its own memory is based on load and store operations upon memory. Shared-memory systems (including distributed shared memory systems) have a single address space and any processor can acquire any data within the memory by load and store. The situation is different for distributed parallel systems. Specialized MPP systems such as the T3E can simulate shared-memory by direct data acquisition from remote memory. But if the parallel code is distributed across a cluster, or across the Net, messages must be sent and received using the protocols for long-distance communication, such as TCP/IP. This requires a ``handshaking'' between nodes of the distributed system. One can think of the method of negotiated communication (e.g MPI), sends and recvs.

MPI is a standard developed for distributed computing across loosely-coupled systems, and therefore incurs a software penalty for negotiating the communication. It is an open industry standard. openMP is another level of parallelization implemented in FMS models to take advantage of the multi-core supercomputer systems.

The message-passing requirements of climate and weather codes can be reduced to a fairly simple minimal set, which is easily implemented in any message-passing API. mpp\_mod provides this API. Features of mpp\_mod include:

1) Simple, minimal API, with free access to underlying API for more complicated stuff.

2) Design toward typical use in climate/weather CFD codes.

3) Performance to be not significantly lower than any native API.

Parallel computing is initially daunting, but it soon becomes second nature, much the way many of us can now write vector code without much effort. The key insight required while reading and writing parallel code is in arriving at a mental grasp of several independent parallel execution streams through the same code. Each variable you examine may have different values for each stream, the processor ID being an obvious example. Subroutines and function calls are particularly subtle, since it is not always obvious from looking at a call what synchronization between execution streams it implies.

It is therefore important to be conscious of the context of a subroutine or function call, and the implied synchronization. There are certain calls here (e.g mpp\_declare\_pelist, mpp\_init, mpp\_malloc, mpp\_set\_stack\_size) which must be called by all PEs. There are others which must be called by a subset of PEs (here called a pelist) which must be called by all the PEs in the pelist (e.g mpp\_max, mpp\_sum, mpp\_sync). Still others imply no synchronization at all. I will make every effort to highlight the context of each call in the MPP modules, so that the implicit synchronization is spelt out.

For performance it is necessary to keep synchronization as limited as the algorithm being implemented will allow. For instance, a single message between two processors should only imply synchronization across the PEs in question. A global synchronization (or barrier) is likely to be slow, and is best avoided.

Another reason to use pelists is to run a single program, where different PE subsets work on different portions of the code. A typical example is to assign an ocean model and atmosphere model to different PE subsets, and couple them concurrently instead of running them serially. The MPP module provides the notion of a current pelist, which is set when a group of PEs branch off into a subset. Subsequent calls that omit the pelist optional argument (seen below in many of the individual calls) assume that the implied synchronization is across the current pelist. The calls mpp\_root\_pe and mpp\_npes also return the values appropriate to the current pelist. The mpp\_set\_current\_pelist call is provided to set the current pelist.

The most commonly used interface defined in mpp\_mod is to send messages (mpp\_send) and receive messages (mpp\_recv). mpp\_send posts a non-blocking send. Mpp\_recv support both blocking and non-blocking communication. For non-blocking communication, mpp\_sync\_self need to be called to ensure safe communication. Code block 1.4 shows a simple example to use mpp\_send and mpp\_recv.

Call mpp\_recv(recv\_buffer, glen=msgsize, from\_pe=from\_pe, block=.false.)

Call mpp\_send(send\_buffer,plen=msgsize, to\_pe=to\_pe)

Call mpp\_sync\_self(check=EVENT\_RECV) ! ensure the receiving is completed

! Unpack data here

Call mpp\_sync\_self(check=EVEN\_SEND) ! ensure the sending is completed

Code block 1.4: example of using mpp\_send and mpp\_recv

Mpp\_mod provides interface to do performance analysis. It prints out running time of codes surrounded by interface mpp\_clock\_begin and mpp\_clock\_end. Each clock is initialized through interface mpp\_clock\_id. Code block 1.5 provides an example to track the running time of routine update\_ocean\_model.

id\_ocean = mpp\_clock\_id(‘Ocean’)

Call mpp\_clock\_begin(id\_ocean)

Call update\_ocean\_model

Call mpp\_clock\_end(id\_ocean)

Code block 1.5: example to track running time

3.1 Mpp\_domains\_mod

mpp\_domains\_mod is a set of simple calls for domain decomposition and domain updates on rectilinear grids. Scalable implementations of finite-difference codes are generally based on decomposing the model domain into subdomains that are distributed among processors. These domains will then be obliged to exchange data at their boundaries. If data dependencies are merely nearest-neighbour, or may need to acquire information from the global domain if there are extended data dependencies, as in the spectral transform. The domain decomposition is a key operation in the development of parallel codes.

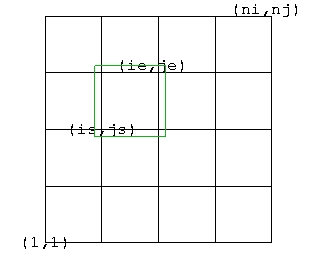


Figure 1.6: domain decomposition on 16 processors with layout (4,4). On each processor, the global domain is (1:ni,1:nj), the computing domain is (is:ie,js:je).

It is assumed that domain decomposition will mainly be in 2 horizontal dimensions, which will in general be the two fastest-varying indices. There is a separate implementation of 1D decomposition on the fastest-varying index, and 1D decomposition on the second index, treated as a special case of 2D decomposition, is also possible. We define domain as the grid associated with a task. We define the compute domain as the set of grid points that are computed by a task, and the data domain as the set of points that are required by the task for the calculation. There can in general be more than 1 task per PE, though often the number of domains is the same as the processor count. We define the global domain as the global computational domain of the entire model (i.e, the same as the computational domain if run on a single processor). 2D domains are defined using a derived type domain2D. Figure 1.6 provides an example of simple domain decomposition.

mpp\_domains\_mod provides a domain decomposition and domain update API for rectilinear grids, built on top of the mpp\_mod API for message passing. Features of mpp\_domains\_mod include:

* Simple, minimal API, with free access to underlying API for more complicated stuff.
* Design toward typical use in climate/weather CFD codes.

The most commonly used interfaces in mpp\_domains\_mod are mpp\_define\_domains and mpp\_update\_domains. Mpp\_define\_domains is used to setup a domain decomposition. Mpp\_updates\_domains is used to perform a halo update of a domain-decomposed array on each processor. Code block (1.3) defined a domain for a double cyclic region with grid size (nx,ny). Layout could be defined through mpp\_define\_layout or picked by user (like layout = (4,4) in Figure 1.6). mpp\_get\_compute\_domains retrieves the compute domain size and mpp\_get\_data\_domains retries the data domain size. The computation is done on compute domain and the data is located on data domain. Code block 1.3 provides both scalar and vector version of mpp\_update\_domains.

call mpp\_define\_layout( (/1,nx,1,ny/), npes, layout )

call mpp\_define\_domains( (/1,nx,1,ny), npes, domain, xhalo=1, yhalo=1, &

xflags=CYCLIC\_GLOBAL\_DOMAIN, yflags=CYCLIC\_GLOBAL\_DOMAIN )

call mpp\_get\_compute\_domains(domain, isc, iec, jsc, jec)

call mpp\_get\_data\_domains(domain, isd, ied, jsd, jed)

allocate( x(isd:ied,jsd:jed) )

call mpp\_update\_domains(x, domain)

call mpp\_update\_domains(x, y, domain, gridtype=BGRID\_NE)

Code block 1.6: example of using mpp\_domains API

3.3 mpp\_io\_mod

mpp\_io\_mod, is a set of simple calls for parallel I/O on distributed systems. It is geared toward the writing of data in netCDF format.

In massively parallel environments, an often difficult problem is the reading and writing of data to files on disk. mpp\_io\_mod is an attempt at a simple API encompassing a certain variety of the I/O tasks that will be required. It does not attempt to be an all-encompassing standard such as MPI, however, it can be implemented in MPI if so desired. It is equally simple to add parallel I/O capability to mpp\_io\_mod based on vendor-specific APIs while providing a layer of insulation for user codes.

The mpp\_io\_mod parallel I/O API built on top of the mpp\_domains\_mod and mpp\_mod API for domain decomposition and message passing. Features of mpp\_io\_mod include:

* Simple, minimal API, with free access to underlying API for more complicated stuff.
* Self-describing files: comprehensive header information (metadata) in the file itself.
* Strong focus on performance of parallel write: the climate models for which it is designed typically read a minimal amount of data (typically at the beginning of the run); but on the other hand, tend to write copious amounts of data during the run. An interface for reading is also supplied, but its performance has not yet been optimized.
* Integrated netCDF capability: netCDF is a data format widely used in the climate/weather modeling community. netCDF is considered the principal medium of data storage for mpp\_io\_mod. But I provide a raw unformatted fortran I/O capability in case netCDF is not an option, either due to unavailability, inappropriateness, or poor performance.

The internal representation of the data being written out is assumed be the default real type, which can be 4 or 8-byte. Time data is always written as 8-bytes to avoid overflow on climatic time scales in units of seconds.

The I/O activity critical to performance in the models for which mpp\_io\_mod is designed is typically the writing of large datasets on a model grid volume produced at intervals during a run. Consider a 3D grid volume, where model arrays are stored as (i,j,k). The domain decomposition is typically along i or j: thus to store data to disk as a global volume, the distributed chunks of data have to be seen as non-contiguous. If we attempt to have all processors write this data into a single file, performance can be seriously compromised because of the data reordering that will be required. Possible options are to have one processor acquire all the data and write it out, or to have all the processors write independent files, which are recombined offline. The performance can also be compromised when running on large processor count.

These three modes of operation are described in the mpp\_io\_mod terminology in terms of two parameters, threading and fileset, as follows: Single-threaded I/O: a single processor acquires all the data and writes it out. Multi-threaded, single-fileset I/O: many processors write to a single file. Multi-threaded, multi-fileset I/O: many processors write to independent files. This is also called distributed I/O.

The distributed I/O also has performance issue when high resolution model runs on large processor count (> 1000). The latest approach is to define an IO domain with io\_layout to divide all the processors into a certain number of groups. The first processor of each group acquires all the data on its group and writes to a single file (The file will be different for each group). This could be a balance of number of files to be written and data size to be acquired and written out. The layout of io\_domain will depend on the layout of grid domain. Figure 1.7 gives an example of the IO domain.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 18 | 19 | 20 | 21 | 22 | 23 |
| 12 | 13 | 14 | 15 | 16 | 17 |
| 6 | 7 | 8 | 9 | 10 | 11 |
| 0 | 1 | 2 | 3 | 4 | 5 |

Figure 1.7: Use IO domain to control IO. The layout of the grid domain is (6,4) and the io\_layout is (2,2). Processors 0 to 23 are divided into 4 groups: (0,1,2,6,7,8), (3,4,5,9,10,11), (12,13,14,18,19,20) and (15,16,17,21,22,23). 4 processors(0,3,12,15) write to 4 independent files.

Mpp\_io\_mod provides interface to open and close files, read metadata and data from files, write metadata and data to files. Code block 1.7 gives an example on how to read data using mpp\_io\_mod interfaces. Code block 1.8 gives an example on how to write data using mpp\_io\_mod interfaces.

call mpp\_open( unit, “foo.nc”, action=MPP\_RDONLY, &

form=MPP\_NETCDF, threading=MPP\_MULTI, fileset=MPP\_SINGLE )

call mpp\_get\_info( unit, ndim, nvar, natt, ntime )

allocate( vars(nvar)

call mpp\_get\_fields ( unit, vars(:) )

call mpp\_read( unit, vars(1), domain, data)

call mpp\_close(unit)

Code block 1.7: example to read data

call mpp\_open( unit, “foo.nc”, action=MPP\_OVERWR, &

form=MPP\_NETCDF, threading=MPP\_MULTI, fileset=MPP\_MULTI )

call mpp\_write\_meta( unit, id\_x, 'X', 'km', 'X distance', 'X', domain=xdom, data=(/(i,i=1,nx)/) )

call mpp\_write\_meta( unit, id\_y, 'Y', 'km', 'Y distance', 'Y', domain=ydom, data=(/(i,i=1,ny)/) )

call mpp\_write\_meta( unit, id\_f, (/x,y/), 'Data', 'metres', 'Random data', pack=1 )

call mpp\_write( unit, x )

call mpp\_write( unit, y )

call mpp\_write( unit, data, domain, data)

call mpp\_close(unit)

Code block 1.8: example to write data

3.4 data\_override

At runtime, the atmosphere, ice, land, and/or ocean component models can be disabled through namelist flags coupler nml: do\_atmos, do\_ice, do\_land, do\_ocean. FMS infrastructure provides method (interface data\_override) to override the component boundary fields in running time. Data\_override\_init must be called (only need to be called once) before calling data\_override. The boundary fields could be overridden by a constant value or with data from a specified NetCDF file. Spatial and temporal interpolation will be performed if necessary to convert data to model’s grid and time. This information is provided through the data override table. Before using data\_override, a data\_table must be created with the following entries: gridname, fieldname\_code, fieldname\_file, file\_name, ongrid and factor. For instance, the sea surface temperature (SST) to the ocean ( from the ice model ) can be overridden with data from a sea surface temperature observation data with following table entry,

cat > data\_table <<EOF

‘OCN’, ‘sst\_obs’, ‘sst’, ‘INPUT/sst\_obs.nc’, .false., 0.01

EOF

Table 1.2: example of data\_override table.

‘OCN’ --- identifies the target component model.

‘sst\_obs’ --- corresponds to the name of boundary field in calling data\_override.

‘sst’ --- is the name of the field as it exists in the NetCDF data file.

‘INPUT/sst\_obs.nc’ ---- is the name of the file containing SST data

.false. --- data is on the model grid ( true) or off-grid (false)

0.01 --- scale factor.

Interface data\_override is used to override the boundary fields in the model code. Data\_override\_init (only need to be called once) must be called before calling any data\_override. Code block 1.9 gives examples to override sea surface temperature and wind stress to the ocean.

call data\_overide\_init()

call data\_override('OCN','sst\_obs',sst,Time)

call data\_override('OCN','u\_flux',u\_flux,Time)

call data\_override('OCN','v\_flux',v\_flux,Time)

Code block 1.9: source code of using data\_override.

**3.5 diag\_manager**

diag\_manager\_mod is a set of simple calls for parallel diagnostics on distributed systems. It is geared toward the writing of data in netCDF format. diag\_manager\_mod provides a convenient set of interfaces for writing data to disk. It is built upon the parallel I/O interface of FMS codes. Run-time specification of diagnostics is input through the diagnostics table.

Diag\_manager has the ability to output from 0-D array (scalars) to 3-D arrays and to output time averages of fields that have time dependent mask. By default, a field is output in its global grid, it is now possible to output a field in a region specified by user in the diag\_table. To check if the diag table is set up correctly, user should set diag\_manager\_nml variable debug\_diag\_manager=true. Then the content of diag\_table is printed in standard output. A field will be written out to disk only when it is registered in the model code and is specified in the diag\_table.

Code block 1.10 gives an example on how to use interfaces of diag\_manager. Diag\_manager\_init should be called first to initialize diag\_manager\_mode. Diag\_manager\_end needs to be called last to complete writing data to the disk. Both diag\_manager\_init and diag\_manager\_end only need to be called once in a program.

Call diag\_manager\_init

id\_axis(1) = diag\_axis\_init('lon', lon, 'degrees\_E', 'x', long\_name='longitude',Domain2=Domain)

id\_axis(2) = diag\_axis\_init('lat', lat, 'degrees\_N', ‘y', long\_name='latitude', Domain2=Domain)

id\_eta\_t = register\_diag\_field ('ocean\_model', 'eta\_t', id\_axis, time, 'surface height on T cells',

'meter', missing\_value=-10.0, range=(/-10.0,10.0/))

used = send\_data (id\_sea\_level, eta\_t , time, rmask=mask)

call diag\_manager\_end

Usage of diag\_manager includes the following steps:

Code block 1.10: Example to use diag\_manager\_mod interfaces.

A diag\_table consists of three sections: global section, file section and field section. Global section is the first two lines of the table contain the experiment title and base date. The base date is the reference time used for the time units. The base date must be greater than or equal to the model start date. The date consists of six space-separated integers: year, month, day, hour, minute, and second.

Each line of file section specifies one output file. It contains 6 fields -"file\_name", output\_freq, "output\_freq\_units", format,"time\_units", "time\_long\_name". “file\_name” is the name of the file to be written out. When “output\_freq” = 0, output frequency at every time step; when “output\_freq” = -1, output frequency at end of run; when “output\_freq” > 0, output frequency in “output\_freq\_units”. output\_freq\_units is the units used for output frequency. When “format” = 1, the output file will be in NetCDF format and is the only format supported. “time\_units” is the units used to lable the time axis. “time\_long\_name”

Each line in field section specifies one field to be written out. It contains 8 fields – "module\_name", "field\_name", "output\_name", "file\_name" "time\_sampling", “time\_avg”, "other\_opts", “packing”. “module\_name” is the component model name, e.g. “ocean\_model”. “field\_name” is the field name when registered the field through register\_diag\_field. “output\_name” is the name of the field in the output file. “file\_name” is the name of the file to be written out and is associated with the “file\_name” is the file section. “time\_avg” could be .true. (time averaging) or .false. (instant). When “packing”=1, the output is in double precision; when “packing”=2, the output is in float precision, when “packing”=4, the output is packed into 16-bit integer.

Table 1.3 gives an example of diagnostics table.

cat >> diag\_table <<EOF

CM2.1U\_Control-1990\_E1.M\_3A

1990 1 1 0 0 0

"ocean\_month", 1, "months", 1, "days", "time",

“ocean\_daily", 24, "hours", 1, "days", "time",

"ocean\_model","eta\_t","eta\_t","ocean\_month","all",.true.,"none",2

"ocean\_model","eta\_t","eta\_t","ocean\_daily","all", .false.,"none",2

EOF

\* Format (should always be 1)

\* time axis unit

\* time axis name

\* frequency for creating new file

\* unit for creating new file

Table 1.3: example of diag\_table.

In summary, this chapter has presented a review of Flexible modeling system. It includes the key features of how coupling is performed in the GFDL Flexible Modeling System and important modules in the FMS infrastructure. A standard coupling interface with slots for atmosphere, land surface, ocean surface, and ocean components is coupled along with a surface boundary layer component on an exchange grid. Components live within a single executable, but can be scheduled serially or concurrently with others. The code has been shown to be scalable to O(1000) processors, with fast surface processes coupling every atmospheric timestep (typically about 15 min) and slow processes coupling every ocean timestep (typically 1 hour). Coupling is conservative to up to second-order accuracy.

FMS infrastructure provides interfaces to support parallelization, input/output, diagnostics and boundary forcing data for the component models and coupler. This infrastructure should largely insulate model developers from machine-specific details. It also insulate FMS users from Message-passing library ( MPI ) and scientific data format library ( NetCDF). It provides interfaces commonly used in different component models. It standardize the diagnostics output for all the component models.