Earth System Modeling Framework

ESMF Reference Manual for Fortran

Version 8.9.0 beta snapshot

ESMF Joint Specification Team: V. Balaji, Byron Boville, Samson Cheung, Tom Clune, Nancy Collins, Tony Craig, Carlos Cruz, Arlindo da Silva, Cecelia DeLuca, Rosalinda de Fainchtein, Rocky Dunlap, Brian Eaton, Steve Goldhaber, Bob Hallberg, Tom Henderson, Chris Hill, Mark Iredell, Joseph Jacob, Rob Jacob, Phil Jones, Brian Kauffman, Erik Kluzek, Ben Koziol, Jay Larson, Peggy Li, Fei Liu, John Michalakes, Raffaele Montuoro, Sylvia Murphy, David Neckels, Ryan O Kuinghttons, Bob Oehmke, Chuck Panaccione, Daniel Rosen, Jim Rosinski, Mathew Rothstein, Bill Sacks, Kathy Saint, Will Sawyer, Earl Schwab, Shepard Smithline, Walter Spector, Don Stark, Max Suarez, Spencer Swift, Gerhard Theurich, Atanas Trayanov, Silverio Vasquez, Jon Wolfe, Weiyu Yang, Mike Young, Leonid Zaslavsky

January 16, 2025

---

http://www.earthsystemmodeling.org

Acknowledgements

The ESMF software is based on the contributions of a broad community. Below are the software packages that are included in ESMF or strongly influenced our design. We'd like to express our gratitude to the developers of these codes for access to their software as well as their ideas and advice.

- Parallel I/O (PIO) developers at NCAR and DOE Laboratories for their excellent work on this package and their help in making it work with ESMF
- The Spherical Coordinate Remapping and Interpolation Package (SCRIP) from Los Alamos, which informed the design of our regridding functionality
- The Model Coupling Toolkit (MCT) from Argonne National Laboratory, on which we based our sparse matrix multiply approach to general regridding
- The Inpack configuration attributes package from NASA Goddard, which was adapted for use in ESMF by members of NASA Global Modeling and Assimilation group
- The Flexible Modeling System (FMS) package from GFDL and the Goddard Earth Modeling System (GEMS) from NASA Goddard, both of which provided inspiration for the overall ESMF architecture
- The Common Component Architecture (CCA) effort within the Department of Energy, from which we drew many ideas about how to design components
- The Vector Signal Image Processing Library (VSIPL) and its predecessors, which informed many aspects of our design, and the radar system software design group at Lincoln Laboratory
- The Portable, Extensible Toolkit for Scientific Computation (PETSc) package from Argonne National Laboratories, on which we based our initial makefile system
- The Community Climate System Model (CCSM) and Weather Research and Forecasting (WRF) modeling groups at NCAR, who have provided valuable feedback on the design and implementation of the framework

Contents

Part I

ESMF Overview

1 What is the Earth System Modeling Framework?

The Earth System Modeling Framework (ESMF) is a suite of software tools for developing high-performance, multi-component Earth science modeling applications. Such applications may include a few or dozens of components representing atmospheric, oceanic, terrestrial, or other physical domains, and their constituent processes (dynamical, chemical, biological, etc.). Often these components are developed by different groups independently, and must be "coupled" together using software that transfers and transforms data among the components in order to form functional simulations.

ESMF supports the development of these complex applications in a number of ways. It introduces a set of simple, consistent component interfaces that apply to all types of components, including couplers themselves. These interfaces expose in an obvious way the inputs and outputs of each component. It offers a variety of data structures for transferring data between components, and libraries for regridding, time advancement, and other common modeling functions. Finally, it provides a growing set of tools for using metadata to describe components and their input and output fields. This capability is important because components that are self-describing can be integrated more easily into automated workflows, model and dataset distribution and analysis portals, and other emerging "semantically enabled" computational environments.

ESMF is not a single Earth system model into which all components must fit, and its distribution doesn't contain any scientific code. Rather it provides a way of structuring components so that they can be used in many different user-written applications and contexts with minimal code modification, and so they can be coupled together in new configurations with relative ease. The idea is to create many components across a broad community, and so to encourage new collaborations and combinations.

ESMF offers the flexibility needed by this diverse user base. It is tested nightly on more than two dozen platform/compiler combinations; can be run on one processor or thousands; supports shared and distributed memory programming models and a hybrid model; can run components sequentially (on all the same processors) or concurrently (on mutually exclusive processors); and supports single executable or multiple executable modes.

ESMF's generality and breadth of function can make it daunting for the novice user. To help users navigate the software, we try to apply consistent names and behavior throughout and to provide many examples. The large-scale structure of the software is straightforward. The utilities and data structures for building modeling components are called the ESMF *infrastructure*. The coupling interfaces and drivers are called the *superstructure*. User code sits between these two layers, making calls to the infrastructure libraries underneath and being scheduled and synchronized by the superstructure above. The configuration resembles a sandwich, as shown in Figure 1.

ESMF users may choose to extensively rewrite their codes to take advantage of the ESMF infrastructure, or they may decide to simply wrap their components in the ESMF superstructure in order to utilize framework coupling services. Either way, we encourage users to contact our support team if questions arise about how to best use the software, or how to structure their application. ESMF is more than software; it's a group of people dedicated to realizing the vision of a collaborative model development community that spans institutional and national bounds.

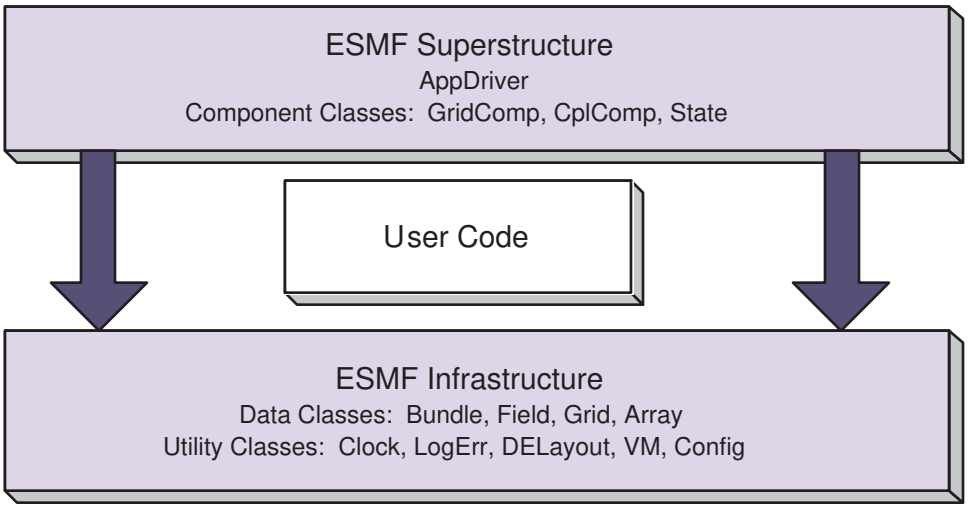
2 The ESMF Reference Manual for Fortran

ESMF has a complete set of Fortran interfaces and some C interfaces. This *ESMF Reference Manual* is a listing of ESMF interfaces for Fortran.[1]

Interfaces are grouped by class. A class is comprised of the data and methods for a specific concept like a physical field. Superstructure classes are listed first in this *Manual*, followed by infrastructure classes.

[1] Since the customer base for it is small, we have not yet prepared a comprehensive reference manual for C.

Figure 1: Schematic of the ESMF "sandwich" architecture. The framework consists of two parts, an upper level **superstructure** layer and a lower level **infrastructure** layer. User code is sandwiched between these two layers.



The major classes in the ESMF superstructure are Components, which usually represent large pieces of functionality such as atmosphere and ocean models, and States, which are the data structures used to transfer data between Components. There are both data structures and utilities in the ESMF infrastructure. Data structures include multi-dimensional Arrays, Fields that are comprised of an Array and a Grid, and collections of Arrays and Fields called ArrayBundles and FieldBundles, respectively. There are utility libraries for data decomposition and communications, time management, logging and error handling, and application configuration.

3 How to Contact User Support and Find Additional Information

The ESMF team can answer questions about the interfaces presented in this document. For user support, please contact esmf_support@ucar.edu.

The website, http://www.earthsystemmodeling.org, provide more information of the ESMF project as a whole. The website includes release notes and known bugs for each version of the framework, supported platforms, project history, values, and metrics, related projects, the ESMF management structure, and more. The *ESMF User's Guide* contains build and installation instructions, an overview of the ESMF system and a description of how its classes interrelate (this version of the document corresponds to the last public version of the framework). Also available on the ESMF website is the *ESMF Developer's Guide* that details ESMF procedures and conventions.

4 How to Submit Comments, Bug Reports, and Feature Requests

We welcome input on any aspect of the ESMF project. Send questions and comments to esmf_support@ucar.edu.

5 Conventions

5.1 Typeface and Diagram Conventions

The following conventions for fonts and capitalization are used in this and other ESMF documents.

Style	Meaning	Example
italics	documents	*ESMF Reference Manual*
`courier`	code fragments	`ESMF_TRUE`
`courier()`	ESMF method name	`ESMF_FieldGet()`
boldface	first definitions	An **address space** is ...
boldface	web links and tabs	**Developers** tab on the website
Capitals	ESMF class name	DataMap

ESMF class names frequently coincide with words commonly used within the Earth system domain (field, grid, component, array, etc.) The convention we adopt in this manual is that if a word is used in the context of an ESMF class name it is capitalized, and if the word is used in a more general context it remains in lower case. We would write, for example, that an ESMF Field class represents a physical field.

Diagrams are drawn using the Unified Modeling Language (UML). UML is a visual tool that can illustrate the structure of classes, define relationships between classes, and describe sequences of actions. A reader interested in more detail can refer to a text such as *The Unified Modeling Language Reference Manual.* [19]

5.2 Method Name and Argument Conventions

Method names begin with `ESMF_`, followed by the class name, followed by the name of the operation being performed. Each new word is capitalized. Although Fortran interfaces are not case-sensitive, we use case to help parse multi-word names.

For method arguments that are multi-word, the first word is lower case and subsequent words begin with upper case. ESMF class names (including typed flags) are an exception. When multi-word class names appear in argument lists, all letters after the first are lower case. The first letter is lower case if the class is the first word in the argument and upper case otherwise. For example, in an argument list the DELayout class name may appear as `delayout` or `srcDelayout`.

Most Fortran calls in the ESMF are subroutines, with any returned values passed through the interface. For the sake of convenience, some ESMF calls are written as functions.

A typical ESMF call looks like this:

```
call ESMF_<ClassName><Operation>(classname, firstArgument,
           secondArgument, ..., rc)
```

where
`<ClassName>` is the class name,
`<Operation>` is the name of the action to be performed,
`classname` is a variable of the derived type associated with the class,
the `arg*` arguments are whatever other variables are required for the operation,
and `rc` is a return code.

6 The ESMF Application Programming Interface

The ESMF Application Programming Interface (API) is based on the object-oriented programming concept of a **class**. A class is a software construct that is used for grouping a set of related variables together with the subroutines and functions that operate on them. We use classes in ESMF because they help to organize the code, and often make it easier to maintain and understand. A particular instance of a class is called an **object**. For example, Field is an ESMF class. An actual Field called `temperature` is an object. That is about as far as we will go into software engineering terminology.

The Fortran interface is implemented so that the variables associated with a class are stored in a derived type. For example, an `ESMF_Field` derived type stores the data array, grid information, and metadata associated with a physical field. The derived type for each class is stored in a Fortran module, and the operations associated with each class are defined as module procedures. We use the Fortran features of generic functions and optional arguments extensively to simplify our interfaces.

The modules for ESMF are bundled together and can be accessed with a single `USE` statement, `USE ESMF`.

6.1 Standard Methods and Interface Rules

ESMF defines a set of standard methods and interface rules that hold across the entire API. These are:

- `ESMF_<Class>Create()` and `ESMF_<Class>Destroy()`, for creating and destroying objects of ESMF classes that require internal memory management (- called ESMF deep classes). The `ESMF_<Class>Create()` method allocates memory for the object itself and for internal variables, and initializes variables where appropriate. It is always written as a Fortran function that returns a derived type instance of the class, i.e. an object.
- `ESMF_<Class>Set()` and `ESMF_<Class>Get()`, for setting and retrieving a particular item or flag. In general, these methods are overloaded for all cases where the item can be manipulated as a name/value pair. If identifying the item requires more than a name, or if the class is of sufficient complexity that overloading in this way would result in an overwhelming number of options, we define specific `ESMF_<Class>Set<Something>()` and `ESMF_<Class>Get<Something>()` interfaces.
- `ESMF_<Class>Add()`, `ESMF_<Class>AddReplace()`, `ESMF_<Class>Remove()`, and `ESMF_<Class>Replace()`, for manipulating objects of ESMF container classes - such as `ESMF_State` and `ESMF_FieldBundle`. For example, the `ESMF_FieldBundleAdd()` method adds another Field to an existing FieldBundle object.
- `ESMF_<Class>Print()`, for printing the contents of an object to standard out. This method is mainly intended for debugging.
- `ESMF_<Class>ReadRestart()` and `ESMF_<Class>WriteRestart()`, for saving the contents of a class and restoring it exactly. Read and write restart methods have not yet been implemented for most ESMF classes, so where necessary the user needs to write restart values themselves.
- `ESMF_<Class>Validate()`, for determining whether a class is internally consistent. For example, `ESMF_FieldValidate()` validates the internal consistency of a Field object.

6.2 Deep and Shallow Classes

ESMF contains two types of classes.

Deep classes require ESMF_<Class>Create() and ESMF_<Class>Destroy() calls. They involve memory allocation, take significant time to set up (due to memory management) and should not be created in a time-critical portion of code. Deep objects persist even after the method in which they were created has returned. Most classes in ESMF, including GridComp, CplComp, State, Fields, FieldBundles, Arrays, ArrayBundles, Grids, and Clocks, fall into this category.

Shallow classes do not possess ESMF_<Class>Create() and ESMF_<Class>Destroy() calls. They are simply declared and their values set using an ESMF_<Class>Set() call. Examples of shallow classes are Time, TimeInterval, and ArraySpec. Shallow classes do not take long to set up and can be declared and set within a time-critical code segment. Shallow objects stop existing when execution goes out of the declaring scope.

An exception to this is when a shallow object, such as a Time, is stored in a deep object such as a Clock. The deep Clock object then becomes the declaring scope of the Time object, persisting in memory. The Time object is deallocated with the ESMF_ClockDestroy() call.

See Section 9, Overall Design and Implementation Notes, for a brief discussion of deep and shallow classes from an implementation perspective. For an in-depth look at the design and inter-language issues related to deep and shallow classes, see the *ESMF Implementation Report*.

6.3 Aliases and Named Aliases

Deep objects, i.e. instances of ESMF deep classes created by the appropriate ESMF_<Class>Create(), can be used with the standard assignment (=), equality (==), and not equal (/=) operators.

The assignment

```
deep2 = deep1
```

makes deep2 an **alias** of deep1, meaning that both variables reference the same deep allocation in memory. Many aliases of the same deep object can be created.

All the aliases of a deep object are equivalent. In particular, there is no distinction between the variable on the left hand side of the actual ESMF_<Class>Create() call, and any aliases created from it. All actions taken on any of the aliases of a deep object affect the deep object, and thus all other aliases.

The equality and not equal operators for deep objects are implemented as simple alias checks. For a more general comparison of two distinct deep objects, a deep class might provide the ESMF_<Class>Match() method.

ESMF provides the concept of a **named alias**. A named alias behaves just like an alias in all aspects, except when it comes to setting and getting the *name* of the deep object it is associated with. While regular aliases all access the same name string in the actual deep object, a named alias keeps its private name string. This allows the same deep object to be known under a different name in different contexts.

The assignment

```
deep2 = ESMF_NamedAlias(deep1)
```

makes deep2 a **named alias** of deep1. Any *name* changes on deep2 only affect deep2. However, the *name* retrieved from deep1, or from any regular aliases created from deep1, is unaffected.

Notice that aliases generated from a named alias are again named aliases. This is true even when using the regular assignment operator with a named alias on the right hand side. Named aliases own their unique name string that cannot be accessed or altered through any other alias.

6.4 Special Methods

The following are special methods which, in one case, are required by any application using ESMF, and in the other case must be called by any application that is using ESMF Components.

- `ESMF_Initialize()` and `ESMF_Finalize()` are required methods that must bracket the use of ESMF within an application. They manage the resources required to run ESMF and shut it down gracefully. ESMF does not support restarts in the same executable, i.e. `ESMF_Initialize()` should not be called after `ESMF_Finalize()`.
- `ESMF_<Type>CompInitialize()`, `ESMF_<Type>CompRun()`, and `ESMF_<Type>CompFinalize()` are component methods that are used at the highest level within ESMF. `<Type>` may be `<Grid>`, for Gridded Components such as oceans or atmospheres, or `<Cpl>`, for Coupler Components that are used to connect them. The content of these methods is not part of the ESMF. Instead the methods call into associated subroutines within user code.

6.5 The ESMF Data Hierarchy

The ESMF API is organized around a hierarchy of classes that contain model data. The operations that are performed on model data, such as regridding, redistribution, and halo updates, are methods of these classes.

The main data classes in ESMF, in order of increasing complexity, are:

- **Array** An ESMF Array is a distributed, multi-dimensional array that can carry information such as its type, kind, rank, and associated halo widths. It contains a reference to a native Fortran array.
- **ArrayBundle** An ArrayBundle is a collection of Arrays, not necessarily distributed in the same manner. It is useful for performing collective data operations and communications.
- **Field** A Field represents a physical scalar or vector field. It contains a reference to an Array along with grid information and metadata.
- **FieldBundle** A FieldBundle is a collection of Fields discretized on the same grid. The staggering of data points may be different for different Fields within a FieldBundle. Like the ArrayBundle, it is useful for performing collective data operations and communications.
- **State** A State represents the collection of data that a Component either requires to run (an Import State) or can make available to other Components (an Export State). States may contain references to Arrays, ArrayBundles, Fields, FieldBundles, or other States.
- **Component** A Component is a piece of software with a distinct function. ESMF currently recognizes two types of Components. Components that represent a physical domain or process, such as an atmospheric model, are called Gridded Components since they are usually discretized on an underlying grid. The Components responsible for regridding and transferring data between Gridded Components are called Coupler Components. Each Component is associated with an Import and an Export State. Components can be nested so that simpler Components are contained within more complex ones.

Underlying these data classes are native language arrays. ESMF allows you to reference an existing Fortran array to an ESMF Array or Field so that ESMF data classes can be readily introduced into existing code. You can perform communication operations directly on Fortran arrays through the VM class, which serves as a unifying wrapper for distributed and shared memory communication libraries.

6.6 ESMF Spatial Classes

Like the hierarchy of model data classes, ranging from the simple to the complex, ESMF is organized around a hierarchy of classes that represent different spaces associated with a computation. Each of these spaces can be manipulated, in order to give the user control over how a computation is executed. For Earth system models, this hierarchy starts with the address space associated with the computer and extends to the physical region described by the application. The main spatial classes in ESMF, from those closest to the machine to those closest to the application, are:

- The **Virtual Machine**, or **VM** The ESMF VM is an abstraction of a parallel computing environment that encompasses both shared and distributed memory, single and multi-core systems. Its primary purpose is resource allocation and management. Each Component runs in its own VM, using the resources it defines. The elements of a VM are **Persistent Execution Threads**, or **PETs**, that are executing in **Virtual Address Spaces**, or **VASs**. A simple case is one in which every PET is associated with a single MPI process. In this case every PET is executing in its own private VAS. If Components are nested, the parent component allocates a subset of its PETs to its children. The children have some flexibility, subject to the constraints of the computing environment, to decide how they want to use the resources associated with the PETs they've received.
- **DELayout** A DELayout represents a data decomposition (we also refer to this as a distribution). Its basic elements are **Decomposition Elements**, or **DEs**. A DELayout associates a set of DEs with the PETs in a VM. DEs are not necessarily one-to-one with PETs. For cache blocking, or user-managed multi-threading, more DEs than PETs may be defined. Fewer DEs than PETs may also be defined if an application requires it.
- **DistGrid** A DistGrid represents the index space associated with a grid. It is a useful abstraction because often a full specification of grid coordinates is not necessary to define data communication patterns. The DistGrid contains information about the sequence and connectivity of data points, which is sufficient information for many operations. Arrays are defined on DistGrids.
- **Array** An Array defines how the index space described in the DistGrid is associated with the VAS of each PET. This association considers the type, kind and rank of the indexed data. Fields are defined on Arrays.
- **Grid** A Grid is an abstraction for a logically rectangular region in physical space. It associates a coordinate system, a set of coordinates, and a topology to a collection of grid cells. Grids in ESMF are comprised of DistGrids plus additional coordinate information.
- **Mesh** A Mesh provides an abstraction for an unstructured grid. Coordinate information is set in nodes, which represent vertices or corners. Together the nodes establish the boundaries of mesh elements or cells.
- **LocStream** A LocStream is an abstraction for a set of unstructured data points without any topological relationship to each other.
- **Field** A Field may contain more dimensions than the Grid that it is discretized on. For example, for convenience during integration, a user may want to define a single Field object that holds snapshots of data at multiple times. Fields also keep track of the stagger location of a Field data point within its associated Grid cell.

6.7 ESMF Maps

In order to define how the index spaces of the spatial classes relate to each other, we require either implicit rules (in which case the relationship between spaces is defined by default), or special Map arrays that allow the user to specify the desired association. The form of the specification is usually that the position of the array element carries information about the first object, and the value of the array element carries information about the second object. ESMF includes a `distGridToArrayMap`, a `gridToFieldMap`, a `distGridToGridMap`, and others.

6.8 ESMF Specification Classes

It can be useful to make small packets of descriptive parameters. ESMF has one of these:

- **ArraySpec**, for storing the specifics, such as type/kind/rank, of an array.

6.9 ESMF Utility Classes

There are a number of utilities in ESMF that can be used independently. These are:

- **Attributes**, for storing metadata about Fields, FieldBundles, States, and other classes.
- **TimeMgr**, for calendar, time, clock and alarm functions.
- **LogErr**, for logging and error handling.
- **Config**, for creating resource files that can replace namelists as a consistent way of setting configuration parameters.

7 Integrating ESMF into Applications

Depending on the requirements of the application, the user may want to begin integrating ESMF in either a top-down or bottom-up manner. In the top-down approach, tools at the superstructure level are used to help reorganize and structure the interactions among large-scale components in the application. It is appropriate when interoperability is a primary concern; for example, when several different versions or implementations of components are going to be swapped in, or a particular component is going to be used in multiple contexts. Another reason for deciding on a top-down approach is that the application contains legacy code that for some reason (e.g., intertwined functions, very large, highly performance-tuned, resource limitations) there is little motivation to fully restructure. The superstructure can usually be incorporated into such applications in a way that is non-intrusive.

In the bottom-up approach, the user selects desired utilities (data communications, calendar management, performance profiling, logging and error handling, etc.) from the ESMF infrastructure and either writes new code using them, introduces them into existing code, or replaces the functionality in existing code with them. This makes sense when maximizing code reuse and minimizing maintenance costs is a goal. There may be a specific need for functionality or the component writer may be starting from scratch. The calendar management utility is a popular place to start.

7.1 Using the ESMF Superstructure

The following is a typical set of steps involved in adopting the ESMF superstructure. The first two tasks, which occur before an ESMF call is ever made, have the potential to be the most difficult and time-consuming. They are the work of splitting an application into components and ensuring that each component has well-defined stages of execution. ESMF aside, this sort of code structure helps to promote application clarity and maintainability, and the effort put into it is likely to be a good investment.

1. Decide how to organize the application as discrete Gridded and Coupler Components. This might involve reorganizing code so that individual components are cleanly separated and their interactions consist of a minimal number of data exchanges.

2. Divide the code for each component into initialize, run, and finalize methods. These methods can be multi-phase, e.g., `init_1, init_2`.

3. Pack any data that will be transferred between components into ESMF Import and Export State data structures. This is done by first wrapping model data in either ESMF Arrays or Fields. Arrays are simpler to create and use than Fields, but carry less information and have a more limited range of operations. These Arrays and Fields are then added to Import and Export States. They may be packed into ArrayBundles or FieldBundles first, for more efficient communications. Metadata describing the model data can also be added. At the end of this step, the data to be transferred between components will be in a compact and largely self-describing form.

4. Pack time information into ESMF time management data structures.

5. Using code templates provided in the ESMF distribution, create ESMF Gridded and Coupler Components to represent each component in the user code.

6. Write a set services routine that sets ESMF entry points for each user component's initialize, run, and finalize methods.

7. Run the application using an ESMF Application Driver.

8 Overall Rules and Behavior

8.1 Return Code Handling

All ESMF methods pass a *return code* back to the caller via the `rc` argument. If no errors are encountered during the method execution, a value of `ESMF_SUCCESS` is returned. Otherwise one of the predefined error codes is returned to the caller. See the appendix, section 56, for a full list of the ESMF error return codes.

Any code calling an ESMF method must check the return code. If `rc` is not equal to `ESMF_SUCCESS`, the calling code is expected to break out of its execution and pass the `rc` to the next level up. All ESMF errors are to be handled as *fatal*, i.e. the calling code must *bail-on-all-errors*.

ESMF provides a number of methods, described under section 49, that make implementation of the bail-on-all-errors stategy more convenient. Consistent use of these methods will ensure that a full back trace is generated in the ESMF log output whenever an error condition is triggered.

Note that in ESMF requesting not present information, e.g. via a `Get()` method, will trigger an error condition. Combined with the bail-on-all-errors strategy this has the advantage of producing an error trace pointing to the earliest location in the code that attempts to access unavailable information. In cases where the calling side is able to handle the presence or absence of certain pieces of of information, the code first must query for the resepctive `isPresent` argument. If this argument comes back as `.true.` it is safe to query for the actual information.

8.2 Local and Global Views and Associated Conventions

ESMF data objects such as Fields are distributed over DEs, with each DE getting a portion of the data. Depending on the task, a local or global view of the object may be preferable. In a local view, data indices start with the first element on the DE and end with the last element on the same DE. In a global view, there is an assumed or specified order to the set of DEs over which the object is distributed. Data indices start with the first element on the first DE, and continue across all the elements in the sequence of DEs. The last data index represents the number of elements in the entire object. The DistGrid provides the mapping between local and global data indices.

The convention in ESMF is that entities with a global view have no prefix. Entities with a DE-local (and in some cases, PET-local) view have the prefix "local."

Just as data is distributed over DEs, DEs themselves can be distributed over PETs. This is an advanced feature for users who would like to create multiple local chunks of data, for algorithmic or performance reasons. Local DEs are those DEs that are located on the local PET. Local DE labeling always starts at 0 and goes to localDeCount-1, where localDeCount is the number of DEs on the local PET. Global DE numbers also start at 0 and go to deCount-1. The DELayout class provides the mapping between local and global DE numbers.

8.3 Allocation Rules

The basic rule of allocation and deallocation for the ESMF is: whoever allocates it is responsible for deallocating it.

ESMF methods that allocate their own space for data will deallocate that space when the object is destroyed. Methods which accept a user-allocated buffer, for example `ESMF_FieldCreate()` with the `ESMF_DATACOPY_REFERENCE` flag, will not deallocate that buffer at the time the object is destroyed. The user must deallocate the buffer when all use of it is complete.

Classes such as Fields, FieldBundles, and States may have Arrays, Fields, Grids and FieldBundles created externally and associated with them. These associated items are not destroyed along with the rest of the data object since it is possible for the items to be added to more than one data object at a time (e.g. the same Grid could be part of many Fields). It is the user's responsibility to delete these items when the last use of them is done.

8.4 Assignment, Equality, Copying and Comparing Objects

The equal sign assignment has not been overloaded in ESMF, thus resulting in the standard Fortran behavior. This behavior has been documented as the first entry in the API documentation section for each ESMF class. For deep ESMF objects the assignment results in setting an alias the the same ESMF object in memory. For shallow ESMF objects the assignment is essentially a equivalent to a copy of the object. For deep classes the equality operators have been overloaded to test for the alias condition as a counter part to the assignment behavior. This and the not equal operator are documented following the assignment in the class API documentation sections.

Deep object copies are implemented as a special variant of the `ESMF_<Class>Create()` methods. It takes an existing deep object as one of the required arguments. At this point not all deep classes have `ESMF_<Class>Create()` methods that allow object copy.

Due to the complexity of deep classes there are many aspects when comparing two objects of the same class. ESMF provide `ESMF_<Class>Match()` methods, which are functions that return a class specific match flag. At this point not all deep classes have `ESMF_<Class>Match()` methods that allow deep object comparison.

8.5 Attributes

Attributes are (name, value) pairs, where the name is a character string and the value can be either a single value or list of `integer`, `real`, `double precision`, `logical`, or `character` values. Attributes can be associated with Fields, FieldBundles, and States. Mixed types are not allowed in a single attribute, and all attribute names must be unique within a single object. Attributes are set by name, and can be retrieved either directly by name or by querying for a count of attributes and retrieving names and values by index number.

8.6 Constants

Named constants are used throughout ESMF to specify the values of many arguments with multiple well defined values in a consistent way. These constants are defined by a derived type that follows this pattern:

```
ESMF_<CONSTANT_NAME>_Flag
```

The values of the constant are then specified by this pattern:

```
ESMF_<CONSTANT_NAME>_<VALUE1>
ESMF_<CONSTANT_NAME>_<VALUE2>
ESMF_<CONSTANT_NAME>_<VALUE3>
...
```

A master list of all available constants can be found in section 54.

9 Overall Design and Implementation Notes

1. **Deep and shallow classes.** The deep and shallow classes described in Section 6.2 differ in how and where they are allocated within a multi-language implementation environment. We distinguish between the implementation language, which is the language a method is written in, and the calling language, which is the language that the user application is written in. Deep classes are allocated off the process heap by the implementation language. Shallow classes are allocated off the stack by the calling language.
2. **Base class.** All ESMF classes are built upon a Base class, which holds a small set of system-wide capabilities.

10 Overall Restrictions and Future Work

1. **32-bit integer limitations.** In general, Fortran array bounds should be limited to 2**31-1 elements or less. This is due to the Fortran-95 limitation of returning default sized (e.g., 32 bit) integers for array bound and size inquiries, and consequent ESMF use of default sized integers for holding these values.

Part II

Command Line Tools

The main product delivered by ESMF is the ESMF library that allows application developers to write programs based on the ESMF API. In addition to the programming library, ESMF distributions come with a small set of command line tools (CLT) that are of general interest to the community. These CLTs utilize the ESMF library to implement features such as printing general information about the ESMF installation, or generating regrid weight files. The provided ESMF CLTs are intended to be used as standard command line tools.

The bundled ESMF CLTs are built and installed during the usual ESMF installation process, which is described in detail in the ESMF User's Guide section "Building and Installing the ESMF". After installation, the CLTs will be located in the `ESMF_APPSDIR` directory, which can be found as a Makefile variable in the `esmf.mk` file. The `esmf.mk` file can be found in the `ESMF_INSTALL_LIBDIR` directory after a successful installation. The ESMF User's Guide discusses the `esmf.mk` mechanism to access the bundled CLTs in more detail in section "Using Bundled ESMF Command Line Tools".

The following sections provide in-depth documentation of the bundled ESMF CLTs. In addition, each tool supports the standard `--help` command line argument, providing a brief description of how to invoke the program.

11 ESMF_PrintInfo

11.1 Description

The `ESMF_PrintInfo` command line tool that prints basic information about the ESMF installation to `stdout`.

The command line tool usage is as follows:

```
ESMF_PrintInfo  [--help]

where
  --help     prints a brief usage message


‘
```

12 ESMF_RegridWeightGen

12.1 Description

This section describes the offline regrid weight generation application provided by ESMF (for a description of ESMF regridding in general see Section 24.2). Regridding, also called remapping or interpolation, is the process of changing the grid that underlies data values while preserving qualities of the original data. Different kinds of transformations are appropriate for different problems. Regridding may be needed when communicating data between Earth system model components such as land and atmosphere, or between different data sets to support operations such as visualization.

Regridding can be broken into two stages. The first stage is generation of an interpolation weight matrix that describes how points in the source grid contribute to points in the destination grid. The second stage is the multiplication of values on the source grid by the interpolation weight matrix to produce values on the destination grid. This is implemented as a parallel sparse matrix multiplication.

There are two options for accessing ESMF regridding functionality: integrated and offline. Integrated regridding is a process whereby interpolation weights are generated via subroutine calls during the execution of the user's code. The integrated regridding can also perform the parallel sparse matrix multiplication. In other words, ESMF integrated regridding allows a user to perform the whole process of interpolation within their code. For a further description of ESMF integrated regridding please see Section **??**. In contrast to integrated regridding, offline regridding is a process whereby interpolation weights are generated by a separate ESMF command line tool, not within the user code. The ESMF offline regridding tool also only generates the interpolation matrix, the user is responsible for reading in this matrix and doing the actual interpolation (multiplication by the sparse matrix) in their code. The rest of this section further describes ESMF offline regridding.

For a discussion of installing and accessing ESMF command line tools such as this one please see the beginning of this part of the reference manual (Section II) or for the quickest approach to just building and accessing the command line tools please refer to the "Building and using bundled ESMF Command Line Tools" Section in the ESMF User's Guide.

This application requires the NetCDF library to read the grid files and to write out the weight files in NetCDF format. To compile ESMF with the NetCDF library, please refer to the "Third Party Libraries" Section in the ESMF User's Guide for more information.

As described above, this tool reads in two grid files and outputs weights for interpolation between the two grids. The input and output files are all in NetCDF format. The grid files can be defined in five different formats: the SCRIP format 12.8.1 as is used as an input to SCRIP [7], the CF convension single-tile grid file 12.8.3 following the CF metadata conventions, the GRIDSPEC Mosaic file 12.8.5 following the proposed GRIDSPEC standard, the ESMF unstructured grid format 12.8.2 or the proposed CF unstructured grid data model (UGRID) 12.8.4. GRIDSPEC is a proposed CF extension for the annotation of complex Earth system grids. In the latest ESMF library, we added support for multi-tile GRIDSPEC Mosaic file with non-overlapping tiles. For UGRID, we support the 2D flexible mesh topology with mixed triangles and quadrilaterals and fully 3D unstructured mesh topology with hexahedrons and tetrahedrons.

The `ESMF_RegridWeightGen` command line tool can detect the type of the input grid files automatically, so the specification of source and destination grid file type arguments is optional. However, these arguments (`-t`, `--src_type` or `--dst_type`) can be provided to override the auto-detection. If not explicitly specified, the rule to determine the file format is the following:

- ESMF_FILEFORMAT_UGRID: a variable with attribute "cf_role" or "standard_name" set to "mesh_topology"
- ESMF_FILEFORMAT_MOSAIC: a variable with attribute "standard_name" set to "grid_mosaic_spec"
- ESMF_FILEFORMAT_TILE: a varilable with attribute "standard_name" set to "grid_tile_spec"
- ESMF_FILEFORMAT_ESMFMESH: variables nodeCoords and elementConn exist
- ESMF_FILEFORMAT_SCRIP: variables grid_corner_lon and grid_corner_lat exist
- ESMF_FILEFORMAT_CFGRID: variables with attributes "degree_north" and "degree_east" (or similar) exist, and other formats aren't matched

This command line tool can do regrid weight generation from a global or regional source grid to a global or regional destination grid. As is true with many global models, this application currently assumes the latitude and longitude

values refer to positions on a perfect sphere, as opposed to a more complex and accurate representation of the Earth's true shape such as would be used in a GIS system. (ESMF's current user base doesn't require this level of detail in representing the Earth's shape, but it could be added in the future if necessary.)

The interpolation weights generated by this application are output to a NetCDF file (specified by the "-w" or "−−weight" keywords). Two type of weight files are supported: the SCRIP format is the same as that generated by SCRIP, see Section 12.9 for a description of the format; and a simple weight file containing only the weights and the source and destination grid indices (In ESMF term, these are the `factorList` and `factorIndexList` generated by the ESMF weight calculation function `ESMF_FieldRegridStore()`. Note that the sequence of the weights in the file can vary with the number of processors used to run the application. This means that two weight files generated by using different numbers of processors can contain exactly the same interpolation matrix, but can appear different in a direct line by line comparison (such as would be done by ncdiff). The interpolation weights can be generated with the bilinear, patch, nearest neighbor, first-order conservative, or second-order conservative methods described in Section 12.3.

Internally this application uses the ESMF public API to generate the interpolation weights. If a source or destination grid is a single tile logically rectangular grid, then `ESMF_GridCreate()` **??** is used to create an ESMF_Grid object. The cell center coordinates of the input grid are put into the center stagger location (`ESMF_STAGGERLOC_CENTER`). In addition, the corner coordinates are also put into the corner stagger location (`ESMF_STAGGERLOC_CORNER`) for conservative regridding. If a grid contains multiple logically rectangular tiles connected with each other by edges, such as a Cubed Sphere grid, the grid can be represented as a multi-tile ESMF_Grid object created using `ESMF_GridCreateMosaic()` **??**. Such a grid is stored in the GRIDSPEC Mosaic and tile file format. 12.8.5 The method `ESMF_MeshCreate()` **??** is used to create an ESMF_Mesh object, if the source or destination grid is an unstructured grid. When making this call, the flag `convert3D` is set to `TRUE` to convert the 2D coordinates into 3D Cartesian coordinates. Internally `ESMF_FieldRegridStore()` is used to generate the weight table and indices table representing the interpolation matrix.

12.2 Regridding Options

The offline regrid weight generation application supports most of the options available in the rest of the ESMF regrid system. The following is a description of these options as relevant to the application. For a more in-depth description see Section 24.2.

12.2.1 Poles

The regridding occurs in 3D to avoid problems with periodicity and with the pole singularity. This application supports four options for handling the pole region (i.e. the empty area above the top row of the source grid or below the bottom row of the source grid). Note that all of these pole options currently only work for logically rectangular grids (i.e. SCRIP format grids with grid_rank=2 or GRIDSPEC single-tile format grids). The first option is to leave the pole region empty ("-p none"), in this case if a destination point lies above or below the top row of the source grid, it will fail to map, yielding an error (unless "-i" is specified). With the next two options, the pole region is handled by constructing an artificial pole in the center of the top and bottom row of grid points and then filling in the region from this pole to the edges of the source grid with triangles. The pole is located at the average of the position of the points surrounding it, but moved outward to be at the same radius as the rest of the points in the grid. The difference between these two artificial pole options is what value is used at the pole. The default pole option ("-p all") sets the value at the pole to be the average of the values of all of the grid points surrounding the pole. For the other option ("-p N"), the user chooses a number N from 1 to the number of source grid points around the pole. For each destination point, the value at the pole is then the average of the N source points surrounding that destination point. For the last pole option ("-p teeth") no artificial pole is constructed, instead the pole region is covered by connecting points across the top and bottom row of the source Grid into triangles. As this makes the top and bottom of the source sphere flat, for

a big enough difference between the size of the source and destination pole regions, this can still result in unmapped destination points. Only pole option "none" is currently supported with the conservative interpolation methods (e.g. "-m conserve") and with the nearest neighbor interpolation methods ("-m nearestdtos" and "-m neareststod").

12.2.2 Masking

Masking is supported for both the logically rectangular grids and the unstructured grids. If the grid file is in the SCRIP format, the variable "grid_imask" is used as the mask. If the value is set to 0 for a grid point, then that point is considered masked out and won't be used in the weights generated by the application. If the grid file is in the ESMF format, the variable "element Mask" is used as the mask. For a grid defined in the GRIDSPEC single-tile or multi-tile grid or in the UGRID convention, there is no mask variable defined. However, a GRIDSPEC single-tile file or a UGRID file may contain both the grid definition and the data. The grid mask is usually constructed using the missing values defined in the data variable. The regridding application provides the argument "−−src_missingvalue" or "−−dst_missingvalue" for users to specify the variable name from where the mask can be constructed.

12.2.3 Extrapolation

The `ESMF_RegridWeightGen` application supports a number of kinds of extrapolation to fill in points not mapped by the regrid method. Please see the sections starting with section 24.2.11 for a description of these methods. When using the application an extrapolation method is specified by using the "−−extrap_method" flag. For the inverse distance weighted average method (nearestidavg), the number of source locations is specified using the "−−extrap_num_src_pnts" flag, and the distance exponent is specified using the "−−extrap_dist_exponent" flag. For the creep fill method (creep), the number of creep levels is specified using the "−−extrap_num_levels" flag.

12.2.4 Unmapped destination points

If a destination point can't be mapped, then the default behavior of the application is to stop with an error. By specifying "-i" or the equivalent "−−ignore_unmapped " the user can cause the application to ignore unmapped destination points. In this case, the output matrix won't contain entries for the unmapped destination points. Note that the unmapped point detection doesn't currently work for nearest destination to source method ("-m nearestdtos"), so when using that method it is as if "-i" is always on.

12.2.5 Line type

Another variation in the regridding supported with spherical grids is **line type**. This is controlled by the "−−line_type" or "-l" flag. This switch allows the user to select the path of the line which connects two points on a sphere surface. This in turn controls the path along which distances are calculated and the shape of the edges that make up a cell. Both of these quantities can influence how interpolation weights are calculated, for example in bilinear interpolation the distances are used to calculate the weights and the cell edges are used to determine to which source cell a destination point should be mapped.

ESMF currently supports two line types: "cartesian" and "greatcircle". The "cartesian" option specifies that the line between two points follows a straight path through the 3D Cartesian space in which the sphere is embedded. Distances are measured along this 3D Cartesian line. Under this option cells are approximated by planes in 3D space, and their boundaries are 3D Cartesian lines between their corner points. The "greatcircle" option specifies that the line between two points follows a great circle path along the sphere surface. (A great circle is the shortest path between two points on a sphere.) Distances are measured along the great circle path. Under this option cells are on the sphere surface, and their boundaries are great circle paths between their corner points.

12.3 Regridding Methods

This regridding application can be used to generate bilinear, patch, nearest neighbor, first-order conservative, or second-order conservative interpolation weights. The following is a description of these interpolation methods as relevant to the offline weight generation application. For a more in-depth description see Section 24.2.

12.3.1 Bilinear

The default interpolation method for the weight generation application is bilinear. The algorithm used by this application to generate the bilinear weights is the standard one found in many textbooks. Each destination point is mapped to a location in the source Mesh, the position of the destination point relative to the source points surrounding it is used to calculate the interpolation weights. A restriction on bilinear interpolation is that ESMF doesn't support self-intersecting cells (e.g. a cell twisted into a bow tie) in the source grid.

12.3.2 Patch

This application can also be used to generate patch interpolation weights. Patch interpolation is the ESMF version of a technique called "patch recovery" commonly used in finite element modeling [16] [14]. It typically results in better approximations to values and derivatives when compared to bilinear interpolation. Patch interpolation works by constructing multiple polynomial patches to represent the data in a source element. For 2D grids, these polynomials are currently 2nd degree 2D polynomials. The interpolated value at the destination point is the weighted average of the values of the patches at that point.

The patch interpolation process works as follows. For each source element containing a destination point we construct a patch for each corner node that makes up the element (e.g. 4 patches for quadrilateral elements, 3 for triangular elements). To construct a polynomial patch for a corner node we gather all the elements around that node. (Note that this means that the patch interpolation weights depends on the source element's nodes, and the nodes of all elements neighboring the source element.) We then use a least squares fitting algorithm to choose the set of coefficients for the polynomial that produces the best fit for the data in the elements. This polynomial will give a value at the destination point that fits the source data in the elements surrounding the corner node. We then repeat this process for each corner node of the source element generating a new polynomial for each set of elements. To calculate the value at the destination point we do a weighted average of the values of each of the corner polynomials evaluated at that point. The weight for a corner's polynomial is the bilinear weight of the destination point with regard to that corner.

The patch method has a larger stencil than the bilinear, for this reason the patch weight matrix can be correspondingly larger than the bilinear matrix (e.g. for a quadrilateral grid the patch matrix is around 4x the size of the bilinear matrix). This can be an issue when performing a regrid weight generation operation close to the memory limit on a machine.

The patch method does not guarantee that after regridding the range of values in the destination field is within the range of values in the source field. For example, if the mininum value in the source field is 0.0, then it's possible that after regridding with the patch method, the destination field will contain values less than 0.0.

This method currently doesn't support self-intersecting cells (e.g. a cell twisted into a bow tie) in the source grid.

12.3.3 Nearest neighbor

The nearest neighbor interpolation options work by associating a point in one set with the closest point in another set. If two points are equally close then the point with the smallest index is arbitrarily used (i.e. the point with that would have the smallest index in the weight matrix). There are two versions of this type of interpolation available in the regrid weight generation application. One of these is the nearest source to destination method ("-m neareststod"). In

this method each destination point is mapped to the closest source point. The other of these is the nearest destination to source method ("-m nearestdtos"). In this method each source point is mapped to the closest destination point. Note that with this method the unmapped destination point detection doesn't work, so no error will be returned even if there are destination points which don't map to any source point.

12.3.4 First-order conservative

The main purpose of this method is to preserve the integral of the field across the interpolation from source to destination. (For a more in-depth description of what this preservation of the integral (i.e. conservation) means please see section 12.4.) In this method the value across each source cell is treated as a constant, so it will typically have a larger interpolation error than the bilinear or patch methods. The first-order method used here is similar to that described in the following paper [18].

By default (or if "−−norm_type dstarea"), the weight w_{ij} for a particular source cell i and destination cell j are calculated as $w_{ij} = f_{ij} * A_{si}/A_{dj}$. In this equation f_{ij} is the fraction of the source cell i contributing to destination cell j, and A_{si} and A_{dj} are the areas of the source and destination cells. If "−−norm_type fracarea", then the weights are further divided by the destination fraction. In other words, in that case $w_{ij} = f_{ij} * A_{si}/(A_{dj} * D_j)$ where D_j is fraction of the destination cell that intersects the unmasked source grid.

To see a description of how the different normalization options affect the values and integrals produced by the conservative methods see section 12.5. For a grid on a sphere this method uses great circle cells, for a description of potential problems with these see 24.2.9.

12.3.5 Second-order conservative

Like the first-order conservative method, this method's main purpose is to preserve the integral of the field across the interpolation from source to destination. (For a more in-depth description of what this preservation of the integral (i.e. conservation) means please see section 12.4.) The difference between the first and second-order conservative methods is that the second-order takes the source gradient into account, so it yields a smoother destination field that typically better matches the source field. This difference between the first and second-order methods is particularly apparent when going from a coarse source grid to a finer destination grid. Another difference is that the second-order method does not guarantee that after regridding the range of values in the destination field is within the range of values in the source field. For example, if the mininum value in the source field is 0.0, then it's possible that after regridding with the second-order method, the destination field will contain values less than 0.0. The implementation of this method is based on the one described in this paper [12].

The weights for second-order are calculated in a similar manner to first-order 12.3.4 with additional weights that take into account the gradient across the source cell.

To see a description of how the different normalization options affect the values and integrals produced by the conservative methods see section 12.5. For a grid on a sphere this method uses great circle cells, for a description of potential problems with these see 24.2.9.

12.4 Conservation

Conservation means that the following equation will hold: $\sum^{all-source-cells}(V_{si} * A'_{si}) = \sum^{all-destination-cells}(V_{dj} * A'_{dj})$, where V is the variable being regridded and A is the area of a cell. The subscripts s and d refer to source and destination values, and the i and j are the source and destination grid cell indices (flattening the arrays to 1 dimension).

There are a couple of options for how the areas (A) in the proceding equation can be calculated. By default, ESMF calculates the areas. For a grid on a sphere, areas are calculated by connecting the corner coordinates of each grid cell (obtained from the grid file) with great circles. For a Cartesian grid, areas are calculated in the typcial manner for 2D polygons. If the user specifies the user area's option ("--user_areas"), then weights will be adjusted so that the equation above will hold for the areas provided in the grid files. In either case, the areas output to the weight file are the ones for which the weights have been adjusted to conserve.

12.5 The effect of normalization options on integrals and values produced by conservative methods

It is important to note that by default (i.e. using destination area normalization) conservative regridding doesn't normalize the interpolation weights by the destination fraction. This means that for a destination grid which only partially overlaps the source grid the destination field which is output from the regrid operation should be divided by the corresponding destination fraction to yield the true interpolated values for cells which are only partially covered by the source grid. The fraction also needs to be included when computing the total source and destination integrals. To include the fraction in the conservative weights, the user can specify the fraction area normalization type. This can be done by specifying "--norm_type fracarea" on the command line.

For weights generated using destination area normalization (either by not specifying any normalization type or by specifying "--norm_type dstarea"), the following pseudo-code shows how to adjust a destination field (`dst_field`) by the destination fraction (`dst_frac`) called `frac_b` in the weight file:

```
for each destination element i
   if (dst_frac(i) not equal to 0.0) then
      dst_field(i)=dst_field(i)/dst_frac(i)
   end if
end for
```

For weights generated using destination area normalization (either by not specifying any normalization type or by specifying "--norm_type dstarea"), the following pseudo-code shows how to compute the total destination integral (`dst_total`) given the destination field values (`dst_field`) resulting from the sparse matrix multiplication of the weights in the weight file by the source field, the destination area (`dst_area`) called `area_b` in the weight file, and the destination fraction (`dst_frac`) called `frac_b` in the weight file. As in the previous paragraph, it also shows how to adjust the destination field (`dst_field`) resulting from the sparse matrix multiplication by the fraction (`dst_frac`) called `frac_b` in the weight file:

```
dst_total=0.0
for each destination element i
   if (dst_frac(i) not equal to 0.0) then
      dst_total=dst_total+dst_field(i)*dst_area(i)
      dst_field(i)=dst_field(i)/dst_frac(i)
      ! If mass computed here after dst_field adjust, would need to be:
      ! dst_total=dst_total+dst_field(i)*dst_area(i)*dst_frac(i)
   end if
end for
```

For weights generated using fraction area normalization (set by specifying "--norm_type fracarea"), no adjustment of the destination field (`dst_field`) by the destination fraction is necessary. The following pseudo-code shows how to compute the total destination integral (`dst_total`) given the destination field values (`dst_field`) resulting from

the sparse matrix multiplication of the weights in the weight file by the source field, the destination area (dst_area) called area_b in the weight file, and the destination fraction (dst_frac) called frac_b in the weight file:

```
dst_total=0.0
for each destination element i
      dst_total=dst_total+dst_field(i)*dst_area(i)*dst_frac(i)
end for
```

For either normalization type, the following pseudo-code shows how to compute the total source integral (src_total) given the source field values (src_field), the source area (src_area) called area_a in the weight file, and the source fraction (src_frac) called frac_a in the weight file:

```
src_total=0.0
for each source element i
   src_total=src_total+src_field(i)*src_area(i)*src_frac(i)
end for
```

12.6 Usage

The command line arguments are all keyword based. Both the long keyword prefixed with '--' or the one character short keyword prefixed with '-' are supported. The format to run the application is as follows:

```
ESMF_RegridWeightGen
        --source|-s src_grid_filename
        --destination|-d dst_grid_filename
        --weight|-w out_weight_file
        [--method|-m bilinear|patch|nearestdtos|neareststod|conserve|conserve2nd]
        [--pole|-p none|all|teeth|1|2|..]
        [--line_type|-l cartesian|greatcircle]
        [--norm_type dstarea|fracarea]
        [--extrap_method none|neareststod|nearestidavg|nearestd|creep|creepnrstd]
        [--extrap_num_src_pnts <N>]
        [--extrap_dist_exponent <P>]
        [--extrap_num_levels <L>]
        [--ignore_unmapped|-i]
        [--ignore_degenerate]
        [--src_type SCRIP|ESMFMESH|UGRID|CFGRID|GRIDSPEC|MOSAIC|TILE]
        [--dst_type SCRIP|ESMFMESH|UGRID|CFGRID|GRIDSPEC|MOSAIC|TILE]
        [-t SCRIP|ESMFMESH|UGRID|CFGRID|GRIDSPEC|MOSAIC|TILE]
        [-r]
        [--src_regional]
        [--dst_regional]
        [--64bit_offset]
        [--netcdf4]
        [--src_missingvalue var_name]
        [--dst_missingvalue var_name]
        [--src_coordinates lon_name,lat_name]
        [--dst_coordinates lon_name,var_name]
        [--tilefile_path filepath]
```

```
        [--src_loc center|corner]
        [--dst_loc center|corner]
        [--user_areas]
        [--weight_only]
        [--check]
        [--checkFlag]
        [--no_log]
        [--help|-h]
        [--version]
        [-V]

where:
  --source or -s      - a required argument specifying the source grid
                        file name

  --destination or -d - a required argument specifying the destination
                        grid file name

  --weight or -w      - a required argument specifying the output regridding
                        weight file name

  --method or -m      - an optional argument specifying which interpolation
                        method is used. The value can be one of the following:

                        bilinear     - for bilinear interpolation, also the
                                       default method if not specified.
                        patch        - for patch recovery interpolation
                        neareststod  - for nearest source to destination interpolation
                        nearestdtos  - for nearest destination to source interpolation
                        conserve     - for first-order conservative interpolation
                        conserve2nd  - for second-order conservative interpolation

  --pole or -p        - an optional argument indicating how to extrapolate
                        in the pole region.
                        The value can be one of the following:

                        none  - No pole, the source grid ends at the top
                                (and bottom) row of nodes specified in
                                <source grid>.
                        all   - Construct an artificial pole placed in the
                                center of the top (or bottom) row of nodes,
                                but projected onto the sphere formed by the
                                rest of the grid. The value at this pole is
                                the average of all the pole values. This
                                is the default option.

                        teeth - No new pole point is constructed, instead
                                the holes at the poles are filled by
                                constructing triangles across the top and
                                bottom row of the source Grid. This can be
                                useful because no averaging occurs, however,
```

because the top and bottom of the sphere are
now flat, for a big enough mismatch between
the size of the destination and source pole
regions, some destination points may still
not be able to be mapped to the source Grid.

<N> - Construct an artificial pole placed in the
center of the top (or bottom) row of nodes,
but projected onto the sphere formed by the
rest of the grid. The value at this pole is
the average of the N source nodes next to
the pole and surrounding the destination
point (i.e. the value may differ for each
destination point. Here N ranges from 1 to
the number of nodes around the pole.

--line_type
or
-l - an optional argument indicating the type of path
lines (e.g. cell edges) follow on a spherical
surface. The default value depends on the regrid
method. For non-conservative methods the default is
cartesian. For conservative methods the default is
greatcircle.

--norm_type - an optional argument indicating the type of normal-
ization to do when generating conservative weights.
The default value is dstarea.

--extrap_method - an optional argument specifying which extrapolation
method is used to handle unmapped destination locations.
The value can be one of the following:

none - no extrapolation method should be used.
This is the default.

neareststod - nearest source to destination. Each
unmapped destination location is mapped
to the closest source location. This
extrapolation method is not supported with
conservative regrid methods (e.g. conserve).

nearestidavg - inverse distance weighted average.
The value of each unmapped destination location
is the weighted average of the closest N
source locations. The weight is the reciprocal
of the distance of the source from the destination
raised to a power P. All the weights contributing
to one destination point are normalized so that
they sum to 1.0. The user can choose N and P by
using --extrap_num_src_pnts and

```
                                --extrap_dist_exponent, but defaults are
                                also provided. This extrapolation method is not
                                supported with conservative regrid methods
                                (e.g. conserve).

                    nearestd    - nearest mapped destination to
                                unmapped destination. Each
                                unmapped destination location is mapped
                                to the closest mapped destination location. This
                                extrapolation method is not supported with
                                conservative regrid methods (e.g. conserve).

                    creep       - creep fill.
                                Here unmapped destination points are filled by
                                moving values from mapped locations to neighboring
                                unmapped locations. The value filled into a
                                new location is the average of its already filled
                                neighbors' values. This process is repeated for
                                the number of levels indicated by the
                                --extrap_num_levels flag. This extrapolation
                                method is not supported with conservative
                                regrid methods (e.g. conserve).

                    creepnrstd  - creep fill with nearest destination.
                                Here unmapped destination points are filled by
                                first doing a creep fill, and then filling the
                                remaining unmapped points by using
                                the nearest destination method (both of these
                                methods are described in the entries above).
                                This extrapolation method is not supported
                                with conservative regrid methods (e.g. conserve).


--extrap_num_src_pnts - an optional argument specifying how many source points
                        should be used when the extrapolation method is
                        nearestidavg. If not specified, the default is 8.

--extrap_dist_exponent - an optional argument specifying the exponent that
                         the distance should be raised to when the
                         extrapolation method is nearestidavg. If not specified,
                         the default is 2.0.

--extrap_num_levels - an optional argument specifying how many levels should
                      be filled for level based extrapolation methods (e.g. creep).

--ignore_unmapped
       or
       -i         - ignore unmapped destination points. If not specified
                    the default is to stop with an error if an unmapped
                    point is found.
```

```
--ignore_degenerate - ignore degenerate cells in the input grids. If not specified
                      the default is to stop with an error if an degenerate
                      cell is found.

--src_type        - an optional argument specifying the source grid file type.
                    The value can be one of SCRIP, ESMFMESH, UGRID, CFGRID, GRIDSPEC, M
                    If neither --src_type nor -t is given, the source grid file type wi
                    determined automatically. (Usually it is unnecessary to provide --s
                    but it can be specified when the automatic file type determination

--dst_type        - an optional argument specifying the destination grid file type.
                    The value can be one of SCRIP, ESMFMESH, UGRID, CFGRID, GRIDSPEC, M
                    If neither --dst_type nor -t is given, the destination grid file ty
                    determined automatically. (Usually it is unnecessary to provide --d
                    but it can be specified when the automatic file type determination

-t                - an optional argument specifying the file types for both the source
                    and the destination grid files.
                    The value can be one of SCRIP, ESMFMESH, UGRID, CFGRID, GRIDSPEC, M
                    If -t is given, then neither --src_type nor --dst_type can be given

-r                - an optional argument specifying that the source and
                    destination grids are regional grids.  If the argument
                    is not given, the grids are assumed to be global.

--src_regional    - an optional argument specifying that the source is
                    a regional grid and the destination is a global grid.

--dst_regional    - an optional argument specifying that the destination
                    is a regional grid and the source is a global grid.

--64bit_offset    - an optional argument specifying that the weight file
                    will be created in the NetCDF 64-bit offset format
                    to allow variables larger than 2GB.  Note the 64-bit
                    offset format is not supported in the NetCDF version
                    earlier than 3.6.0.  An error message will be generated
                    if this flag is specified while the application is
                    linked with a NetCDF library earlier than 3.6.0.

--netcdf4         - an optional argument specifying that the output weight
                    will be created in the NetCDF4 format.  This option
                    only works with NetCDF library version 4.1 and above
                    that was compiled with the NetCDF4 file format enabled
                    (with HDF5 compression). An error message will be
                    generated if these conditions are not met.

--src_missingvalue - an optional argument that defines the variable name
                     in the source grid file if the file type is either CF Convension
                     single-tile or UGRID.  The regridder will generate a mask using
                     the missing values of the data variable.  The missing
                     value is defined using an attribute called "_FillValue"
```

```
                      or "missing_value".
 --dst_missingvalue - an optional argument that defines the variable name
                      in the destination grid file if the file type is
                      CF Convension single-tile or UGRID.  The regridder will generate a
                      the missing values of the data variable.  The missing
                      value is defined using an attribute called "_FillValue"
                      or "missing_value"

--src_coordinates - an optional argument that defines the longitude and
                    latitude variable names in the source grid file if
                    the file type is CF Convension single-tile.  The variable names are
                    separated by comma.  This argument is required in case
                    there are multiple sets of coordinate variables defined
                    in the file.  Without this argument, the offline regrid
                    application will terminate with an error message when
                    multiple coordinate variables are found in the file.

--dst_coordinates - an optional argument that defines the longitude and
                    latitude variable names in the destination grid file
                    if the file type is CF Convension single-tile.  The variable names
                    separated by comma.  This argument is required in case
                    there are multiple sets of coordinate variables defined
                    in the file.  Without this argument, the offline regrid
                    application will terminate with an error message when
                    multiple coordinate variables are found in the file.

--tilefile_path   - the alternative file path for the tile files when either the source
                    or the destination grid is a GRIDSPEC Mosaic grid.  The path can
                    be either relative or absolute.  If it is relative, it is relative
                    to the working directory.  When specified, the gridlocation variabl
                    defined in the Mosaic file will be ignored.

--src_loc         - an optional argument indicating which part of a source
                    grid cell to use for regridding. Currently, this flag is
                    only required for non-conservative regridding when the
                    source grid is an unstructured grid in ESMF or UGRID format.
                    For all other cases, only the center location is supported.
                    The value can be one of the following:

                    center - Regrid using the center location of each grid cell.

                    corner - Regrid using the corner location of each grid cell.

--dst_loc         - an optional argument indicating which part of a destination
                    grid cell to use for regridding. Currently, this flag is
                    only required for non-conservative regridding when the
                    destination grid is an unstructured grid in ESMF or UGRID format.
                    For all other cases, only the center location is supported.
                    The value can be one of the following:

                    center - Regrid using the center location of each grid cell.
```

```
                    corner - Regrid using the corner location of each grid cell.


  --user_areas      - an optional argument specifying that the conservation
                      is adjusted to hold for the user areas provided in
                      the grid files. If not specified, then the
                      conservation will hold for the ESMF calculated
                      (great circle) areas.
                      Whichever areas the conservation holds for are output
                      to the weight file.

   --weight_only    - an optional argument specifying that the output weight file only
                      contains the weights and the source and destination grid's indices

   --check          - Check that the generated weights produce reasonable
                      regridded fields. This is done by calling ESMF_Regrid()
                      on an analytic source field using the weights generated
                      by this application.  The mean relative error between
                      the destination and analytic field is computed, as well
                      as the relative error between the mass of the source and
                      destination fields in the conservative case.

   --checkFlag      - Turn on more expensive extra error checking during
                      weight generation.

   --no_log         - Turn off the ESMF Log files.  By default, ESMF creates
                      multiple log files, one per PET.

   --help or -h     - Print the usage message and exit.

   --version        - Print ESMF version and license information and exit.

   -V               - Print ESMF version number and exit.
```

12.7 Examples

The example below shows the command to generate a set of conservative interpolation weights between a global SCRIP format source grid file (src.nc) and a global SCRIP format destination grid file (dst.nc). The weights are written into file w.nc. In this case the ESMF library and applications have been compiled using an MPI parallel communication library (e.g. setting ESMF_COMM to openmpi) to enable it to run in parallel. To demonstrate running in parallel the mpirun script is used to run the application in parallel on 4 processors.

```
mpirun -np 4 ./ESMF_RegridWeightGen -s src.nc -d dst.nc -m conserve -w w.nc
```

The next example below shows the command to do the same thing as the previous example except for three changes. The first change is this time the source grid is regional ("`--src_regional`"). The second change is that for this

example bilinear interpolation ("`-m bilinear`") is being used. Because bilinear is the default, we could also omit the "`-m bilinear`". The third change is that in this example some of the destination points are expected to not be found in the source grid, but the user is ok with that and just wants those points to not appear in the weight file instead of causing an error ("`-i`").

```
mpirun -np 4 ./ESMF_RegridWeightGen -i --src_regional -s src.nc -d dst.nc \
               -m bilinear -w w.nc
```

The last example shows how to use the missing values of a data variable to generate the grid mask for a CF Convension single-tile file, how to specify the coordinate variable names using "`--src_coordinates`" and use user defined area for the conservative regridding.

```
mpirun -np 4 ./ESMF_RegridWeightGen -s src.nc -d dst.nc -m conserve \
               -w w.nc --src_missingvalue datavar \
               --src_coordinates lon,lat --user_areas
```

In the above example, "datavar" is the variable name defined in the source grid that will be used to construct the mask using its missing values. In addition, "`lon`" and "`lat`" are the variable names for the longitude and latitude values, respectively.

12.8 Grid File Formats

This section describes the grid file formats supported by ESMF. These are typically used either to describe grids to ESMF_RegridWeightGen or to create grids within ESMF. The following table summarizes the features supported by each of the grid file formats.

Feature	SCRIP	ESMF Unstruct.	CF Grid	UGRID	GRIDSPEC Mosaic
Create an unstructured Mesh	YES	YES	NO	YES	NO
Create a logically-rectangular Grid	YES	NO	YES	NO	YES
Create a multi-tile Grid	NO	NO	NO	NO	YES
2D	YES	YES	YES	YES	YES
3D	NO	YES	NO	YES	NO
Spherical coordinates	YES	YES	YES	YES	YES
Cartesian coordinates	NO	YES	NO	NO	NO
Non-conserv regrid on corners	NO	YES	NO	YES	YES

The rest of this section contains a detailed descriptions of each grid file format along with a simple example of the format.

12.8.1 SCRIP Grid File Format

A SCRIP format grid file is a NetCDF file for describing grids. This format is the same as is used by the SCRIP [7] package, and so grid files which work with that package should also work here. When using the ESMF API, the file format flag `ESMF_FILEFORMAT_SCRIP` can be used to indicate a file in this format.

SCRIP format files are capable of storing either 2D logically rectangular grids or 2D unstructured grids. The basic format for both of these grids is the same and they are distinguished by the value of the `grid_rank` variable. Logically rectangular grids have `grid_rank` set to 2, whereas unstructured grids have this variable set to 1.

The following is a sample header of a logically rectangular grid file:

```
netcdf remap_grid_T42 {
dimensions:
      grid_size = 8192 ;
      grid_corners = 4 ;
      grid_rank = 2 ;

variables:
      int grid_dims(grid_rank) ;
      double grid_center_lat(grid_size) ;
         grid_center_lat:units = "radians";
      double grid_center_lon(grid_size) ;
         grid_center_lon:units = "radians" ;
      int grid_imask(grid_size) ;
         grid_imask:units = "unitless" ;
      double grid_corner_lat(grid_size, grid_corners) ;
         grid_corner_lat:units = "radians" ;
      double grid_corner_lon(grid_size, grid_corners) ;
         grid_corner_lon:units ="radians" ;

// global attributes:
         :title = "T42 Gaussian Grid" ;
}
```

The `grid_size` dimension is the total number of cells in the grid; `grid_rank` refers to the number of dimensions. In this case `grid_rank` is 2 for a 2D logically rectangular grid. The integer array `grid_dims` gives the number of grid cells along each dimension. The number of corners (vertices) in each grid cell is given by `grid_corners`. The grid corner coordinates need to be listed in an order such that the corners are in counterclockwise order. Also, note that if your grid has a variable number of corners on grid cells, then you should set `grid_corners` to be the highest value and use redundant points on cells with fewer corners.

The integer array `grid_imask` is used to mask out grid cells which should not participate in the regridding. The array values should be zero for any points that do not participate in the regridding and one for all other points. Coordinate arrays provide the latitudes and longitudes of cell centers and cell corners. The unit of the coordinates can be either "`radians`" or "`degrees`".

Here is a sample header from a SCRIP unstructured grid file:

```
netcdf ne4np4-pentagons {
dimensions:
      grid_size = 866 ;
      grid_corners = 5 ;
      grid_rank = 1 ;
variables:
      int grid_dims(grid_rank) ;
      double grid_center_lat(grid_size) ;
         grid_center_lat:units = "degrees" ;
```

```
      double grid_center_lon(grid_size) ;
         grid_center_lon:units = "degrees" ;
      double grid_corner_lon(grid_size, grid_corners) ;
         grid_corner_lon:units = "degrees";
         grid_corner_lon:_FillValue = -9999. ;
      double grid_corner_lat(grid_size, grid_corners) ;
         grid_corner_lat:units = "degrees" ;
         grid_corner_lat:_FillValue = -9999. ;
      int grid_imask(grid_size) ;
         grid_imask:_FillValue = -9999. ;
      double grid_area(grid_size) ;
         grid_area:units = "radians^2" ;
         grid_area:long_name = "area weights" ;
}
```

The variables are the same as described above, however, here `grid_rank = 1`. In this format there is no notion of which cells are next to which, so to construct the unstructured mesh the connection between cells is defined by searching for cells with the same corner coordinates. (e.g. the same `grid_corner_lat` and `grid_corner_lon` values).

Both the SCRIP grid file format and the SCRIP weight file format work with the SCRIP 1.4 tools.

12.8.2 ESMF Unstructured Grid File Format (ESMFMESH)

ESMF supports a custom unstructured grid file format for describing meshes. This format is more compatible than the SCRIP format with the methods used to create an ESMF Mesh object, so less conversion needs to be done to create a Mesh. The ESMF format is thus more efficient than SCRIP when used with ESMF codes (e.g. the ESMF_RegridWeightGen application). When using the ESMF API, the file format flag `ESMF_FILEFORMAT_ESMFMESH` can be used to indicate a file in this format.

The following is a sample header in the ESMF format followed by a description:

```
netcdf mesh-esmf {
dimensions:
     nodeCount = 9 ;
     elementCount = 5 ;
     maxNodePElement = 4 ;
     coordDim = 2 ;
variables:
     double nodeCoords(nodeCount, coordDim);
            nodeCoords:units = "degrees" ;
     int elementConn(elementCount, maxNodePElement) ;
            elementConn:long_name = "Node Indices that define the element /
                                     connectivity";
            elementConn:_FillValue = -1 ;
            elementConn:start_index = 1 ;
     byte numElementConn(elementCount) ;
            numElementConn:long_name = "Number of nodes per element" ;
     double centerCoords(elementCount, coordDim) ;
            centerCoords:units = "degrees" ;
     double elementArea(elementCount) ;
```

```
            elementArea:units = "radians^2" ;
            elementArea:long_name = "area weights" ;
     int elementMask(elementCount) ;
            elementMask:_FillValue = -9999. ;
// global attributes:
            :gridType="unstructured";
            :version = "0.9" ;
```

In the ESMF format the NetCDF dimensions have the following meanings. The `nodeCount` dimension is the number of nodes in the mesh. The `elementCount` dimension is the number of elements in the mesh. The `maxNodePElement` dimension is the maximum number of nodes in any element in the mesh. For example, in a mesh containing just triangles, then `maxNodePElement` would be 3. However, if the mesh contained one quadrilateral then `maxNodePElement` would need to be 4. The `coordDim` dimension is the number of dimensions of the points making up the mesh (i.e. the spatial dimension of the mesh). For example, a 2D planar mesh would have `coordDim` equal to 2.

In the ESMF format the NetCDF variables have the following meanings. The `nodeCoords` variable contains the coordinates for each node. `nodeCoords` is a two-dimensional array of dimension `(nodeCount,coordDim)`. For a 2D Grid, `coordDim` is 2 and the grid can be either spherical or Cartesian. If the `units` attribute is either `degrees` or `radians`, it is spherical. `nodeCoords(:,1)` contains the longitude coordinates and `nodeCoords(:,2)` contains the latitude coordinates. If the value of the `units` attribute is `km`, `kilometers` or `meters`, the grid is in 2D Cartesian coordinates. `nodeCoords(:,1)` contains the x coordinates and `nodeCoords(:,2)` contains the y coordinates. The same order applies to `centerCoords`. For a 3D Grid, `coordDim` is 3 and the grid is assumed to be Cartesian. `nodeCoords(:,1)` contains the x coordinates, `nodeCoords(:,2)` contains the y coordinates, and `nodeCoords(:,3)` contains the z coordinates. The same order applies to `centerCoords`. A 2D grid in the Cartesian coordinate can only be regridded into another 2D grid in the Cartesian coordinate.

The `elementConn` variable describes how the nodes are connected together to form each element. For each element, this variable contains a list of indices into the `nodeCoords` variable pointing to the nodes which make up that element. By default, the index is 1-based. It can be changed to 0-based by adding an attribute `start_index` of value 0 to the `elementConn` variable. The order of the indices describing the element is important. The proper order for elements available in an ESMF mesh can be found in Section 33.2.1. The file format does support 2D polygons with more corners than those in that section, but internally these are broken into triangles. For these polygons, the corners should be listed such that they are in counterclockwise order around the element. `elementConn` can be either a 2D array or a 1D array. If it is a 2D array, the second dimension of the `elementConn` variable has to be the size of the largest number of nodes in any element (i.e. `maxNodePElement`), the actual number of nodes in an element is given by the `numElementConn` variable. For a given dimension (i.e. `coordDim`) the number of nodes in the element indicates the element shape. For example in 2D, if `numElementConn` is 4 then the element is a quadrilateral. In 3D, if `numElementConn` is 8 then the element is a hexahedron.

If the grid contains some elements with large number of edges, using a 2D array for `elementConn` could take a lot of space. In that case, `elementConn` can be represented as a 1D array that stores the edges of all the elements continuously. When `elementConn` is a 1D array, the dimension `maxNodePElement` is no longer needed, instead, a new dimension variable `connectionCount` is required to define the size of `elementConn`. The value of `connectionCount` is the sum of all the values in `numElementConn`.

The following is an example grid file using 1D array for `elementConn`:

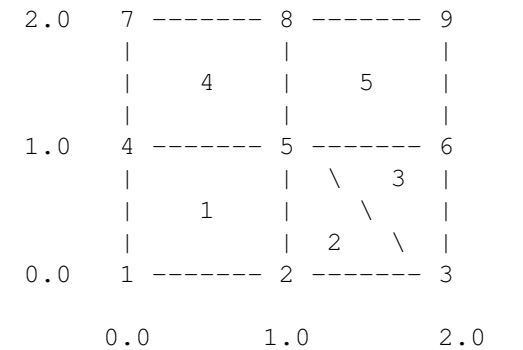
```
netcdf catchments_esmf1 {
dimensions:
        nodeCount = 1824345 ;
        elementCount = 68127 ;
```

```
        connectionCount = 18567179 ;
        coordDim = 2 ;
variables:
        double nodeCoords(nodeCount, coordDim) ;
                nodeCoords:units = ``degrees'' ;
        double centerCoords(elementCount, coordDim) ;
                centerCoords:units = ``degrees'' ;
        int elementConn(connectionCount) ;
                elementConn:polygon_break_value = -8 ;
                elementConn:start_index = 0. ;
        int numElementConn(elementCount) ;
}
```

In some cases, one mesh element may contain multiple polygons and these polygons are separated by a special value defined in the attribute `polygon_break_value`.

The rest of the variables in the format are optional. The `centerCoords` variable gives the coordinates of the center of the corresponding element. This variable is used by ESMF for non-conservative interpolation on the data field residing at the center of the elements. The `elementArea` variable gives the area (or volume in 3D) of the corresponding element. This area is used by ESMF during conservative interpolation. If not specified, ESMF calculates the area (or volume) based on the coordinates of the nodes making up the element. The final variable is the `elementMask` variable. This variable allows the user to specify a mask value for the corresponding element. If the value is 1, then the element is unmasked and if the value is 0 the element is masked. If not specified, ESMF assumes that no elements are masked.

The following is a picture of a small example mesh and a sample ESMF format header using non-optional variables describing that mesh:




```
netcdf mesh-esmf {
dimensions:
        nodeCount = 9 ;
        elementCount = 5 ;
        maxNodePElement = 4 ;
        coordDim = 2 ;
variables:
        double  nodeCoords(nodeCount, coordDim);
                nodeCoords:units = "degrees" ;
```

```
        int elementConn(elementCount, maxNodePElement) ;
                elementConn:long_name = "Node Indices that define the element /
                                         connectivity";
                elementConn:_FillValue = -1 ;
        byte numElementConn(elementCount) ;
                numElementConn:long_name = "Number of nodes per element" ;
// global attributes:
                :gridType="unstructured";
                :version = "0.9" ;
data:
    nodeCoords=
        0.0, 0.0,
        1.0, 0.0,
        2.0, 0.0,
        0.0, 1.0,
        1.0, 1.0,
        2.0, 1.0,
        0.0, 2.0,
        1.0, 2.0,
        2.0, 2.0 ;

    elementConn=
        1, 2, 5,  4,
        2, 3, 5, -1,
        3, 6, 5, -1,
        4, 5, 8,  7,
        5, 6, 9,  8 ;

    numElementConn= 4, 3, 3, 4, 4 ;
}
```

12.8.3 CF Convention Single Tile File Format (CFGRID/GRIDSPEC)

ESMF_RegridWeightGen supports single tile logically rectangular lat/lon grid files that follow the NETCDF CF convention based on CF Metadata Conventions V1.6. When using the ESMF API, the file format flag `ESMF_FILEFORMAT_CFGRID` (or its equivalent deprecated name, `ESMF_FILEFORMAT_GRIDSPEC`) can be used to indicate a file in this format.

An example grid file is shown below. The cell center coordinate variables are determined by the value of its attribute `units`. The longitude variable has the attribute value set to either `degrees_east`, `degree_east`, `degrees_E`, `degree_E`, `degreesE` or `degreeE`. The latitude variable has the attribute value set to `degrees_north`, `degree_north`, `degrees_N`, `degree_N`, `degreesN` or `degreeN`. The latitude and the longitude variables are one-dimensional arrays if the grid is a regular lat/lon grid, two-dimensional arrays if the grid is curvilinear. The bound coordinate variables define the bound or the corner coordinates of a cell. The bound variable name is specified in the `bounds` attribute of the latitude and longitude variables. In the following example, the latitude bound variable is `lat_bnds` and the longitude bound variable is `lon_bnds`. The bound variables are 2D arrays for a regular lat/lon grid and a 3D array for a curvilinear grid. The first dimension of the bound array is 2 for a regular lat/lon grid and 4 for a curvilinear grid. The bound coordinates for a curvilinear grid are defined in counterclockwise order. Since the grid is a regular lat/lon grid, the coordinate variables are 1D and the bound variables are 2D with the first dimension equal

to 2. The bound coordinates will be read in and stored in a ESMF Grid object as the corner stagger coordinates when doing a conservative regrid. In case there are multiple sets of coordinate variables defined in a grid file, the offline regrid application will return an error for duplicate latitude or longitude variables unless "--src_coordinates" or "--src_coordinates" options are used to specify the coordinate variable names to be used in the regrid.

```
netcdf single_tile_grid {
dimensions:
 time = 1 ;
 bound = 2 ;
 lat = 181 ;
 lon = 360 ;
variables:
 double lat(lat) ;
  lat:bounds = "lat_bnds" ;
  lat:units = "degrees_north" ;
  lat:long_name = "latitude" ;
  lat:standard_name = "latitude" ;
 double lat_bnds(lat, bound) ;
 double lon(lon) ;
  lon:bounds = "lon_bnds" ;
  lon:long_name = "longitude" ;
  lon:standard_name = "longitude" ;
  lon:units = "degrees_east" ;
 double lon_bnds(lon, bound) ;
 float so(time, lat, lon) ;
  so:standard_name = "sea_water_salinity" ;
  so:units = "psu" ;
  so:missing_value = 1.e+20f ;
}
```

2D Cartesian coordinates can be supplied in additional to the required longitude/latitude coordinates. They can be used in ESMF to create a grid and used in ESMF_RegridWeightGen. The Cartesian coordinate variables have to include an "axis" attribute with value "X" or "Y". The "units" attribute can be either "m" or "meters" for meters or "km" or "kilometers" for kilometers. When a grid with 2D Cartesian coordinates are used in ESMF_RegridWeightGen, the optional arguments "--src_coordinates" or "--src_coordinates" have to be used to specify the coordinate variable names. A grid with 2D Cartesian coordinates can only be regridded with another grid in 2D Cartesian coordinates. Internally in ESMF, the Cartesian coordinates are all converted into kilometers. Here is an example of the 2D Cartesian coordinates:

```
      double xc(xc) ;
              xc:long_name = "x-coordinate in Cartesian system" ;
              xc:standard_name = "projection_x_coordinate" ;
              xc:axis = "X" ;
              xc:units = "m" ;
      double yc(yc) ;
              yc:long_name = "y-coordinate in Cartesian system" ;
              yc:standard_name = "projection_y_coordinate" ;
              yc:axis = "Y" ;
              yc:units = "m" ;
```

Since a CF convension tile file does not have a way to specify the grid mask, the mask is usually derived by the missing

values stored in a data variable. ESMF_RegridWeightGen provides an option for users to derive the grid mask from a data variable's missing values. The value of the missing value is defined by the variable attribute `missing_value` or `_FillValue`. If the value of the data point is equal to the missing value, the grid mask for that grid point is set to 0, otherwise, it is set to 1. In the following grid, the variable `so` can be used to derive the grid mask. A data variable could be a 2D, 3D or 4D. For example, it may have additional depth and time dimensions. It is assumed that the first and the second dimensions of the data variable should be the longitude and the latitude dimension. ESMF_RegridWeightGen will use the first 2D data values to derive the grid mask.

12.8.4 CF Convention UGRID File Format

ESMF_RegridWeightGen supports NetCDF files that follow the UGRID conventions for unstructured grids.

The UGRID file format is a proposed extension to the CF metadata conventions for the unstructured grid data model. The latest proposal can be found at https://github.com/ugrid-conventions/ugrid-conventions. The proposal is still evolving, the Mesh creation API and ESMF_RegridWeightGen in the current ESMF release is based on UGRID Version 0.9.0 published on October 29, 2013. When using the ESMF API, the file format flag `ESMF_FILEFORMAT_UGRID` can be used to indicate a file in this format.

In the UGRID proposal, a 1D, 2D, or 3D mesh topology can be defined for an unstructured grid. Currently, ESMF supports two types of meshes: (1) the 2D flexible mesh topology where each cell (a.k.a. "face" as defined in the UGRID document) in the mesh is either a triangle or a quadrilateral, and (2) the fully 3D unstructured mesh topology where each cell (a.k.a. "volume" as defined in the UGRID document) in the mesh is either a tetrahedron or a hexahedron. Pyramids and wedges are not currently supported in ESMF, but they can be defined as degenerate hexahedrons. ESMF_RegridWeightGen also supports UGRID 1D network mesh topology in a limited way: A 1D mesh in UGRID can be used as the source grid for nearest neighbor regridding, and as the destination grid for non-conservative regridding.

The main addition of the UGRID extension is a dummy variable that defines the mesh topology. This additional variable has a required attribute `cf_role` with value `"mesh_topology"`. In addition, it has two more required attributes: `topology_dimension` and `node_coordinates`. If it is a 1D mesh, `topology_dimension` is set to 1. If it is a 2D mesh (i.e., `topology_dimension` equals to 2), an additional attribute `face_node_connectivity` is required. If it is a 3D mesh (i.e., `topology_dimension` equals to 3), two additional attributes `volume_node_connectivity` and `volume_shape_type` are required. The value of attribute `node_coordinates` is a list of the names of the node longitude and latitude variables, plus the elevation variable if it is a 3D mesh. The value of attribute `face_node_connectivity` or `volume_node_connectivity` is the variable name that defines the corner node indices for each mesh cell. The additional attribute `volume_shape_type` for the 3D mesh points to a flag variable that specifies the shape type of each cell in the mesh.

Below is a sample 2D mesh called `FVCOM_grid2d`. The dummy mesh topology variable is `fvcom_mesh`. As described above, its `cf_role` attribute has to be `mesh_topology` and the `topology_dimension` attribute has to be 2 for a 2D mesh. It defines the node coordinate variable names to be `lon` and `lat`. It also specifies the face/node connectivity variable name as `nv`.

The variable `nv` is a two-dimensional array that defines the node indices of each face. The first dimension defines the maximal number of nodes for each face. In this example, it is a triangle mesh so the number of nodes per face is 3. Since each face may have a different number of corner nodes, some of the cells may have fewer nodes than the specified dimension. In that case, it is filled with the missing values defined by the attribute `_FillValue`. If `_FillValue` is not defined, the default value is -1. The nodes are in counterclockwise order. An optional attribute `start_index` defines whether the node index is 1-based or 0-based. If `start_index` is not defined, the default node index is 0-based.

The coordinate variables follows the CF metadata convention for coordinates. They are 1D array with attribute

standard_name being either latitude or longitude. The units of the coordinates can be either degrees or radians.

The UGRID files may also contain data variables. The data may be located at the nodes or at the faces. Two additional attributes are introduced in the UGRID extension for the data variables: location and mesh. The location attribute defines where the data is located, it can be either face or node. The mesh attribute defines which mesh topology this variable belongs to since multiple mesh topologies may be defined in one file. The coordinates attribute defined in the CF conventions can also be used to associate the variables to their locations. ESMF checks both location and coordinates attributes to determine where the data variable is defined upon. If both attributes are present, the location attribute takes the precedence. ESMF_RegridWeightGen uses the data variable on the face to derive the element masks for the mesh cell and variable on the node to derive the node masks for the mesh.

When creating a ESMF Mesh from a UGRID file, the user has to provide the mesh topology variable name to ESMF_MeshCreate().

```
netcdf FVCOM_grid2d {
dimensions:
 node = 417642 ;
 nele = 826866 ;
 three = 3 ;
        time  = 1 ;

variables:
// Mesh topology
 int fvcom_mesh;
  fvcom_mesh:cf_role = "mesh_topology" ;
  fvcom_mesh:topology_dimension = 2. ;
  fvcom_mesh:node_coordinates = "lon lat" ;
  fvcom_mesh:face_node_connectivity = "nv" ;
 int nv(nele, three) ;
  nv:standard_name = "face_node_connectivity" ;
  nv:start_index = 1. ;

// Mesh node coordinates
 float lon(node) ;
                lon:standard_name = "longitude" ;
         lon:units = "degrees_east" ;
 float lat(node) ;
                lat:standard_name = "latitude" ;
  lat:units = "degrees_north" ;

// Data variable
 float ua(time, nele) ;
  ua:standard_name = "barotropic_eastward_sea_water_velocity" ;
  ua:missing_value = -999. ;
  ua:location = "face" ;
  ua:mesh = "fvcom_mesh" ;
 float va(time, nele) ;
  va:standard_name = "barotropic_northward_sea_water_velocity" ;
  va:missing_value = -999. ;
  va:location = "face" ;
  va:mesh = "fvcom_mesh" ;
```

}

Following is a sample 3D UGRID file containing hexahedron cells. The dummy mesh topology variable is `fvcom_mesh`. Its `cf_role` attribute has to be `mesh_topology` and `topology_dimension` attribute has to be 3 for a 3D mesh. There are two additional required attributes: `volume_node_connectivity` specifies a variable name that defines the corner indices of the mesh cells and `volume_shape_type` specifies a variable name that defines the type of the mesh cells.

The node coordinates are defined by variables `nodelon`, `nodelat` and `height`. Currently, the units attribute for the height variable is either `kilometers`, `km` or `meters`. The variable `vertids` is a two-dimensional array that defines the corner node indices of each mesh cell. The first dimension defines the maximal number of nodes for each cell. There is only one type of cells in the sample grid, i.e. hexahedrons, so the maximal number of nodes is 8. The node order is defined in 33.2.1. The index can be either 1-based or 0-based and the default is 0-based. Setting an optional attribute `start_index` to 1 changed it to 1-based index scheme. The variable `meshtype` is a one-dimensional integer array that defines the shape type of each cell. Currently, ESMF only supports tetrahedron and hexahedron shapes. There are three attributes in `meshtype`: `flag_range`, `flag_values`, and `flag_meanings` representing the range of the flag values, all the possible flag values, and the meaning of each flag value, respectively. `flag_range` and `flag_values` are either a scalar or an array of integers. `flag_meanings` is a text string containing a list of shape types separated by space. In this example, there is only one shape type, thus, the values of `meshtype` are all 1.

```
netcdf wam_ugrid100_110 {
dimensions:
 nnodes = 78432 ;
 ncells = 66030 ;
 eight = 8 ;
variables:
 int mesh ;
  mesh:cf_role = "mesh_topology" ;
  mesh:topology_dimension = 3. ;
  mesh:node_coordinates = "nodelon nodelat height" ;
  mesh:volume_node_connectivity = "vertids" ;
  mesh:volume_shape_type = "meshtype" ;
 double nodelon(nnodes) ;
  nodelon:standard_name = "longitude" ;
  nodelon:units = "degrees_east" ;
 double nodelat(nnodes) ;
  nodelat:standard_name = "latitude" ;
  nodelat:units = "degrees_north" ;
 double height(nnodes) ;
  height:standard_name = "elevation" ;
  height:units = "kilometers" ;
 int vertids(ncells, eight) ;
  vertids:cf_role = "volume_node_connectivity" ;
  vertids:start_index = 1. ;
 int meshtype(ncells) ;
  meshtype:cf_role = "volume_shape_type" ;
  meshtype:flag_range = 1. ;
  meshtype:flag_values = 1. ;
  meshtype:flag_meanings = "hexahedron" ;
}
```

12.8.5 GRIDSPEC Mosaic File Format

GRIDSPEC is a draft proposal to extend the Climate and Forecast (CF) metadata conventions for the representation of gridded data for Earth System Models. The original GRIDSPEC standard was proposed by V. Balaji and Z. Liang of GFDL (see ref). GRIDSPEC extends the current CF convention to support grid mosaics, i.e., a grid consisting of multiple logically rectangular grid tiles. It also provides a mechanism for storing a grid dataset in multiple files. Therefore, it introduces different types of files, such as a mosaic file that defines the multiple tiles and their connectivity, and a tile file for a single tile grid definition on a so-called "Supergrid" format. When using the ESMF API, the file format flag `ESMF_FILEFORMAT_MOSAIC` can be used to indicate a file in this format.

Following is an example of a mosaic file that defines a 6 tile Cubed Sphere grid:

```
netcdf C48_mosaic {
dimensions:
 ntiles = 6 ;
 ncontact = 12 ;
 string = 255 ;
variables:
 char mosaic(string) ;
  mosaic:standard_name = "grid_mosaic_spec" ;
  mosaic:children = "gridtiles" ;
  mosaic:contact_regions = "contacts" ;
  mosaic:grid_descriptor = "" ;
 char gridlocation(string) ;
 char gridfiles(ntiles, string) ;
 char gridtiles(ntiles, string) ;
 char contacts(ncontact, string) ;
  contacts:standard_name = "grid_contact_spec" ;
  contacts:contact_type = "boundary" ;
  contacts:alignment = "true" ;
  contacts:contact_index = "contact_index" ;
  contacts:orientation = "orient" ;
 char contact_index(ncontact, string) ;
  contact_index:standard_name = "starting_ending_point_index_of_contact" ;

data:

mosaic = "C48_mosaic" ;

gridlocation = "./data/" ;

gridfiles =
  "horizontal_grid.tile1.nc",
  "horizontal_grid.tile2.nc",
  "horizontal_grid.tile3.nc",
  "horizontal_grid.tile4.nc",
  "horizontal_grid.tile5.nc",
  "horizontal_grid.tile6.nc" ;

gridtiles =
  "tile1",
```

```
  "tile2",
  "tile3",
  "tile4",
  "tile5",
  "tile6" ;

contacts =
  "C48_mosaic:tile1::C48_mosaic:tile2",
  "C48_mosaic:tile1::C48_mosaic:tile3",
  "C48_mosaic:tile1::C48_mosaic:tile5",
  "C48_mosaic:tile1::C48_mosaic:tile6",
  "C48_mosaic:tile2::C48_mosaic:tile3",
  "C48_mosaic:tile2::C48_mosaic:tile4",
  "C48_mosaic:tile2::C48_mosaic:tile6",
  "C48_mosaic:tile3::C48_mosaic:tile4",
  "C48_mosaic:tile3::C48_mosaic:tile5",
  "C48_mosaic:tile4::C48_mosaic:tile5",
  "C48_mosaic:tile4::C48_mosaic:tile6",
  "C48_mosaic:tile5::C48_mosaic:tile6" ;

 contact_index =
  "96:96,1:96::1:1,1:96",
  "1:96,96:96::1:1,96:1",
  "1:1,1:96::96:1,96:96",
  "1:96,1:1::1:96,96:96",
  "1:96,96:96::1:96,1:1",
  "96:96,1:96::96:1,1:1",
  "1:96,1:1::96:96,96:1",
  "96:96,1:96::1:1,1:96",
  "1:96,96:96::1:1,96:1",
  "1:96,96:96::1:96,1:1",
  "96:96,1:96::96:1,1:1",
  "96:96,1:96::1:1,1:96" ;
}
```

A GRIDSPEC Mosaic file is identified by a dummy variable with its `standard_name` attribute set to `grid_mosaic_spec`. The `children` attribute of this dummy variable provides the variable name that contains the tile names and the `contact_region` attribute points to the variable name that defines a list of tile pairs that are connected to each other. For a Cubed Sphere grid, there are six tiles and 12 connections. The `contacts` variable, the variable that defines the contact_region has three required attributes: `standard_name`, `contact_type`, and `contact_index`. `startand_name` has to be set to `grid_contact_spec`. `contact_type` can be either `boundary` or `overlap`. Currently, ESMF only supports non-overlapping tiles connected by `boundary`. `contact_index` defines the variable name that contains the information defining how the two adjacent tiles are connected to each other. In the above example, the `contact_index` variable contains 12 entries. Each entry contains the index of four points that defines the two edges that contact to each other from the two neighboring tiles. Assuming the four points are A, B, C, and D. A and B defines the edge of tile 1 and C and D defines the edge of tile 2. A is the same point as C and B is the same as D. (Ai, Aj) is the index for point A. The entry looks like this:

```
  Ai:Bi,Aj:Bj::Ci:Di,Cj:Dj
```

There are two fixed-name variables required in the mosaic file: variable `gridfiles` defines the associated tile

file names and variable `gridlocation` defines the directory path of the tile files. The `gridlocation` can be overwritten with an command line argument `-tilefile_path` in ESMF_RegridWeightGen application.

It is possible to define a single-tile Mosaic file. If there is only one tile in the Mosaic, the `contact_region` attribute in the `grid_mosaic_spec` varilable will be ignored.

Each tile in the Mosaic is a logically rectangular lat/lon grid and is defined in a separate file. The tile file used in the GRIDSPEC Mosaic file defines the coordinates of a so-called `supergrid`. A supergrid contains all the stagger locations in one grid. It contains the corner, edge and center coordinates all in one 2D array. In this example, there are 48 elements in each side of a tile, therefore, the size of the supergrid is 48*2+1=97, i.e. 97x97.

Here is the header of one of the tile files:

```
netcdf horizontal_grid.tile1 {
dimensions:
 string = 255 ;
 nx = 96 ;
 ny = 96 ;
 nxp = 97 ;
 nyp = 97 ;
variables:
 char tile(string) ;
  tile:standard_name = "grid_tile_spec" ;
  tile:geometry = "spherical" ;
  tile:north_pole = "0.0 90.0" ;
  tile:projection = "cube_gnomonic" ;
  tile:discretization = "logically_rectangular" ;
  tile:conformal = "FALSE" ;
 double x(nyp, nxp) ;
  x:standard_name = "geographic_longitude" ;
  x:units = "degree_east" ;
 double y(nyp, nxp) ;
  y:standard_name = "geographic_latitude" ;
  y:units = "degree_north" ;
 double dx(nyp, nx) ;
  dx:standard_name = "grid_edge_x_distance" ;
  dx:units = "meters" ;
 double dy(ny, nxp) ;
  dy:standard_name = "grid_edge_y_distance" ;
  dy:units = "meters" ;
 double area(ny, nx) ;
  area:standard_name = "grid_cell_area" ;
  area:units = "m2" ;
 double angle_dx(nyp, nxp) ;
  angle_dx:standard_name = "grid_vertex_x_angle_WRT_geographic_east" ;
  angle_dx:units = "degrees_east" ;
 double angle_dy(nyp, nxp) ;
  angle_dy:standard_name = "grid_vertex_y_angle_WRT_geographic_north" ;
  angle_dy:units = "degrees_north" ;
 char arcx(string) ;
  arcx:standard_name = "grid_edge_x_arc_type" ;
  arcx:north_pole = "0.0 90.0" ;
```

```
// global attributes:
  :grid_version = "0.2" ;
  :history = "/home/z1l/bin/tools_20091028/make_hgrid --grid_type gnomonic_ed --nlon 96" ;
}
```

The tile file not only defines the coordinates at all staggers, it also has a complete specification of distances, angles, and areas. In ESMF, we only use the `geographic_longitude` and `geographic_latitude` variables and its subsets on the center and corner staggers. ESMF currently supports the Mosaic containing tiles of the same size. A tile can be square or rectangular. For a cubed sphere grid, each tile is a square, i.e. the x and y dimensions are the same.

12.9 Regrid Weight File Format

A regrid weight file is a NetCDF format file containing the information necessary to perform a regridding between two grids. It also optionally contains information about the grids used to compute the regridding. This information is provided to allow applications (e.g. `ESMF_RegridWeightGenCheck`) to independently compute the accuracy of the regridding weights. In some cases, `ESMF_RegridWeightGen` doesn't output the full grid information (e.g. when it's costly to compute, or when the current grid format doesn't support the type of grids used to generate the weights). In that case, the weight file can still be used for regridding, but applications which depend on the grid information may not work.

The following is the header of a sample regridding weight file that describes a bilinear regridding from a logically rectangular 2D grid to a triangular unstructured grid:

```
netcdf t42mpas-bilinear {
dimensions:
        n_a = 8192 ;
        n_b = 20480 ;
        n_s = 42456 ;
        nv_a = 4 ;
        nv_b = 3 ;
        num_wgts = 1 ;
        src_grid_rank = 2 ;
        dst_grid_rank = 1 ;
variables:
        int src_grid_dims(src_grid_rank) ;
        int dst_grid_dims(dst_grid_rank) ;
        double yc_a(n_a) ;
               yc_a:units = "degrees" ;
        double yc_b(n_b) ;
               yc_b:units = "radians" ;
        double xc_a(n_a) ;
               xc_a:units = "degrees" ;
        double xc_b(n_b) ;
               xc_b:units = "radians" ;
        double yv_a(n_a, nv_a) ;
               yv_a:units = "degrees" ;
        double xv_a(n_a, nv_a) ;
               xv_a:units = "degrees" ;
```

```
        double yv_b(n_b, nv_b) ;
               yv_b:units = "radians" ;
        double xv_b(n_b, nv_b) ;
               xv_b:units = "radians" ;
        int mask_a(n_a) ;
               mask_a:units = "unitless" ;
        int mask_b(n_b) ;
               mask_b:units = "unitless" ;
        double area_a(n_a) ;
               area_a:units = "square radians" ;
        double area_b(n_b) ;
               area_b:units = "square radians" ;
        double frac_a(n_a) ;
               frac_a:units = "unitless" ;
        double frac_b(n_b) ;
               frac_b:units = "unitless" ;
        int col(n_s) ;
        int row(n_s) ;
        double S(n_s) ;

// global attributes:
        :title = "ESMF Offline Regridding Weight Generator" ;
        :normalization = "destarea" ;
        :map_method = "Bilinear remapping" ;
        :ESMF_regrid_method = "Bilinear" ;
        :conventions = "NCAR-CSM" ;
        :domain_a = "T42_grid.nc" ;
        :domain_b = "grid-dual.nc" ;
        :grid_file_src = "T42_grid.nc" ;
        :grid_file_dst = "grid-dual.nc" ;
        :ESMF_version = "ESMF_8_2_0_beta_snapshot_05-3-g2193fa3f8a" ;
}
```

The weight file contains four types of information: a description of the source grid, a description of the destination grid, the output of the regrid weight calculation, and global attributes describing the weight file.

12.9.1 Source Grid Description

The variables describing the source grid in the weight file end with the suffix "_a". To be consistent with the original use of this weight file format the grid information is written to the file such that the location being regridded is always the cell center. This means that the grid structure described here may not be identical to that in the source grid file. The full set of these variables may not always be present in the weight file. The following is an explanation of each variable:

n_a The number of source cells.

nv_a The maximum number of corners (i.e. vertices) around a source cell. If a cell has less than the maximum number of corners, then the remaining corner coordinates are repeats of the last valid corner's coordinates.

xc_a The longitude coordinates of the centers of each source cell.

yc_a The latitude coordinates of the centers of each source cell.

xv_a The longitude coordinates of the corners of each source cell.

yv_a The latitude coordinates of the corners of each source cell.

mask_a The mask for each source cell. A value of 0, indicates that the cell is masked.

area_a The area of each source cell. This quantity is either from the source grid file or calculated by `ESMF_RegridWeightGen`. When a non-conservative regridding method (e.g. bilinear) is used, the area is set to 0.0.

src_grid_rank The number of dimensions of the source grid. Currently this can only be 1 or 2. Where 1 indicates an unstructured grid and 2 indicates a 2D logically rectangular grid.

src_grid_dims The number of cells along each dimension of the source grid. For unstructured grids this is equal to the number of cells in the grid.

12.9.2 Destination Grid Description

The variables describing the destination grid in the weight file end with the suffix "_b". To be consistent with the original use of this weight file format the grid information is written to the file such that the location being regridded is always the cell center. This means that the grid structure described here may not be identical to that in the destination grid file. The full set of these variables may not always be present in the weight file. The following is an explanation of each variable:

n_b The number of destination cells.

nv_b The maximum number of corners (i.e. vertices) around a destination cell. If a cell has less than the maximum number of corners, then the remaining corner coordinates are repeats of the last valid corner's coordinates.

xc_b The longitude coordinates of the centers of each destination cell.

yc_b The latitude coordinates of the centers of each destination cell.

xv_b The longitude coordinates of the corners of each destination cell.

yv_b The latitude coordinates of the corners of each destination cell.

mask_b The mask for each destination cell. A value of 0, indicates that the cell is masked.

area_b The area of each destination cell. This quantity is either from the destination grid file or calculated by `ESMF_RegridWeightGen`. When a non-conservative regridding method (e.g. bilinear) is used, the area is set to 0.0.

dst_grid_rank The number of dimensions of the destination grid. Currently this can only be 1 or 2. Where 1 indicates an unstructured grid and 2 indicates a 2D logically rectangular grid.

dst_grid_dims The number of cells along each dimension of the destination grid. For unstructured grids this is equal to the number of cells in the grid.

12.9.3 Regrid Calculation Output

The following is an explanation of the variables containing the output of the regridding calculation:

n_s The number of entries in the regridding matrix.

col The position in the source grid for each entry in the regridding matrix.

row The position in the destination grid for each entry in the weight matrix.

S The weight for each entry in the regridding matrix.

frac_a When a conservative regridding method is used, this contains the fraction of each source cell that participated in the regridding. When a non-conservative regridding method is used, this array is set to 0.0.

frac_b When a conservative regridding method is used, this contains the fraction of each destination cell that participated in the regridding. When a non-conservative regridding method is used, this array is set to 1.0 where the point participated in the regridding (i.e. was within the unmasked source grid), and 0.0 otherwise.

The following code shows how to apply the weights in the weight file to interpolate a source field (`src_field`) defined over the source grid to a destination field (`dst_field`) defined over the destination grid. The variables `n_s`, `n_b`, `row`, `col`, and `S` are from the weight file.

```
! Initialize destination field to 0.0
do i=1, n_b
  dst_field(i)=0.0
enddo

! Apply weights
do i=1, n_s
  dst_field(row(i))=dst_field(row(i))+S(i)*src_field(col(i))
enddo
```

If the first-order conservative interpolation method is specified ("-m conserve") then the destination field may need to be adjusted by the destination fraction (`frac_b`). This should be done if the normalization type is "dstarea" and if the destination grid extends outside the unmasked source grid. If it isn't known if the destination extends outside the source, then it doesn't hurt to apply the destination fraction. (If it doesn't extend outside, then the fraction will be 1.0 everywhere anyway.) The following code shows how to adjust an already interpolated destination field (`dst_field`) by the destination fraction. The variables `n_b`, and `frac_b` are from the weight file:

```
! Adjust destination field by fraction
do i=1, n_b
  if (frac_b(i) .ne. 0.0) then
     dst_field(i)=dst_field(i)/frac_b(i)
  endif
enddo
```

12.9.4 Weight File Description Attributes

The following is an explanation of the global attributes describing the weight file:

title Always set to "ESMF Offline Regridding Weight Generator" when generated by `ESMF_RegridWeightGen`.

normalization The normalization type used to compute conservative regridding weights. The options for this are described in section 12.3.4 which contains a description of the conservative regridding method.

map_method An indication of the mapping method which is constrained by the original use of this format. In some cases the method specified here will differ from the actual regridding method used, for example weights generated with the "patch" method will have this attribute set to "Bilinear remapping".

ESMF_regrid_method The ESMF regridding method used to generate the weight file.

conventions The set of conventions that the weight file follows. Currently only "NCAR-CSM" is supported.

domain_a The source grid file name.

domain_b The destination grid file name.

grid_file_src The source grid file name.

grid_file_dst The destination grid file name.

ESMF_version The version of ESMF used to generate the weight file.

12.9.5 Weight Only Weight File

In the current ESMF distribution, a new simplified weight file option `-weight_only` is added to `ESMF_RegridWeightGen`. The simple weight file contains only a subset of the Regrid Calculation Output defined in 12.9.3, i.e. the weights `S`, the source grid indices `col` and destination grid indices `row`. The dimension of these three variables is `n_s`.

12.10 ESMF_RegridWeightGenCheck

The ESMF_RegridWeightGen application is used in the ESMF_RegridWeightGenCheck external demo to generate interpolation weights. These weights are then tested by using them for a regridding operation and then comparing them against an analytic function on the destination grid. This external demo is also used to regression test ESMF regridding, and it is run nightly on over 150 combinations of structured and unstructured, regional and global grids, and regridding methods.

13 ESMF_Regrid

13.1 Description

This section describes the file-based regridding command line tool provided by ESMF (for a description of ESMF regridding in general see Section 24.2). Regridding, also called remapping or interpolation, is the process of changing the grid that underlies data values while preserving qualities of the original data. Different kinds of transformations are appropriate for different problems. Regridding may be needed when communicating data between Earth system model components such as land and atmosphere, or between different data sets to support operations such as visualization.

Regridding can be broken into two stages. The first stage is generation of an interpolation weight matrix that describes how points in the source grid contribute to points in the destination grid. The second stage is the multiplication of values on the source grid by the interpolation weight matrix to produce values on the destination grid. This is implemented as a parallel sparse matrix multiplication.

The `ESMF_RegridWeightGen` command line tool described in Section 12 performs the first stage of the regridding process - generate the interpolation weight matrix. This tool not only calculates the interpolation weights, it also applies the weights to a list of variables stored in the source grid file and produces the interpolated values on the destination grid. The interpolated output variable is written out to the destination grid file. This tool supports three CF compliant file formats: the CF Single Tile grid file format(12.8.3) for a logically rectangular grid, the UGRID file format(12.8.4) for unstructured grid and the GRIDSPEC Mosaic file format(12.8.5) for cubed-sphere grid. For the GRIDSPEC Mosaic file format, the data are stored in separate data files, one file per tile. The SCRIP format(12.8.1) and the ESMF unstructured grid format(12.8.2) are not supported because there is no way to define a variable field using these two formats. Currently, the tool only works with 2D grids, the support for the 3D grid will be made available in the future release. The variable array can be up to four dimensions. The variable type is currently limited to single or double precision real numbers. The support for other data types, such as integer or short will be added in the future release.

The user interface of this tool is greatly simplified from ESMF_RegridWeightGen. User only needs to provide two input file names, the source and the destination variable names and the regrid method. The tool will figure out the type of the grid file automatically based on the attributes of the variable. If the variable has a coordinates attribute, the grid file is a GRIDSPEC file and the value of the coordinates defines the longitude and latitude variable's names. For example, following is a simple GRIDSPEC file with a variable named PSL and coordinate variables named lon and lat.

```
netcdf simple_gridspec {
dimensions:
      lat = 192 ;
      lon = 288 ;
variables:
      float PSL(lat, lon) ;
         PSL:time = 50. ;
         PSL:units = "Pa" ;
         PSL:long_name = "Sea level pressure" ;
         PSL:cell_method = "time: mean" ;
         PSL:coordinates = "lon lat" ;
      double lat(lat) ;
         lat:long_name = "latitude" ;
         lat:units = "degrees_north" ;
      double lon(lon) ;
         lon:long_name = "longitude" ;
         lon:units = "degrees_east" ;
}
```

If the variable has a mesh attribute and a location attribute, the grid file is in UGRID format(12.8.4). The value of mesh attribute is the name of a dummy variable that defines the mesh topology. If the application performs a conservative regridding, the value of the location attribute has to be face, otherwise, it has to be node. This is because ESMF only supports non-conservative regridding on the data stored at the nodes of a ESMF_Mesh object, and conservative regridding on the data stored at the cells of a ESMF_Mesh object.

Here is an example 2D UGRID file:

```
netcdf simple_ugrid {
dimensions:
      node = 4176 ;
      nele = 8268 ;
      three = 3 ;
      time  = 2 ;
variables:
      float lon(node) ;
         lon:units = "degrees_east" ;
      float lat(node) ;
         lat:units = "degrees_north" ;
      float lonc(nele) ;
         lonc:units = "degrees_east" ;
      float latc(nele) ;
         latc:units = "degrees_north" ;
      int nv(nele, three) ;
         nv:standard_name = "face_node_connectivity" ;
```

```
         nv:start_index = 1. ;
      float zeta(time, node) ;
         zeta:standard_name = "sea_surface_height_above_geoid" ;
         zeta:_FillValue = -999. ;
         zeta:location = "node" ;
         zeta:mesh = "fvcom_mesh" ;
      float ua(time, nele) ;
         ua:standard_name = "barotropic_eastward_sea_water_velocity" ;
         ua:_FillValue = -999. ;
         ua:location = "face" ;
         ua:mesh = "fvcom_mesh" ;
      float va(time, nele) ;
         va:standard_name = "barotropic_northward_sea_water_velocity" ;
         va:_FillValue = -999. ;
         va:location = "face" ;
         va:mesh = "fvcom_mesh" ;
      int fvcom_mesh(node) ;
         fvcom_mesh:cf_role = "mesh_topology" ;
         fvcom_mesh:dimension = 2. ;
         fvcom_mesh:locations = "face node" ;
         fvcom_mesh:node_coordinates = "lon lat" ;
         fvcom_mesh:face_coordinates = "lonc latc" ;
         fvcom_mesh:face_node_connectivity = "nv" ;
}
```

There are three variables defined in the above UGRID file - `zeta` on the node of the mesh, `ua` and `va` on the face of the mesh. All three variables have one extra time dimension.

The GRIDSPEC MOSAIC file(12.8.5) can be identified by a dummy variable with `standard_name` attribute set to `grid_mosaic_spec`. The data for a GRIDSPEC Mosaic file are stored in separate files, one tile per file. The name of the data file is not specified in the mosaic file. Therefore, additional optional argument `-srcdatafile` or `-dstdatafile` is required to provide the prefix of the datafile. The datafile is also a CF compliant NetCDF file. The complete name of the datafile is constructed by appending the tilename (defined in the Mosaic file in a variable specified by the `children` attribute of the dummy variable). For instance, if the prefix of the datafile is `mosaicdata`, then the datafile names are `mosaicdata.tile1.nc`, `mosaicdata.tile2.nc`, etc... using the mosaic file example in 12.8.5. The path of the datafile is defined by `gridlocation` variable, similar to the tile files. To overwrite it, an optional argument `tilefile_path` can be specified.

Following is an example GRIDSPEC MOSAIC datafile:

```
netcdf mosaictest.tile1 {
dimensions:
     grid_yt = 48 ;
     grid_xt = 48 ;
     time = UNLIMITED ; // (12 currently)
variables:
     float area_land(grid_yt, grid_xt) ;
        area_land:long_name = "area in the grid cell" ;
        area_land:units = "m2" ;
     float evap_land(time, grid_yt, grid_xt) ;
        evap_land:long_name = "vapor flux up from land" ;
        evap_land:units = "kg/(m2 s)" ;
```

```
        evap_land:coordinates = "geolon_t geolat_t" ;
    double geolat_t(grid_yt, grid_xt) ;
        geolat_t:long_name = "latitude of grid cell centers" ;
        geolat_t:units = "degrees_N" ;
    double geolon_t(grid_yt, grid_xt) ;
        geolon_t:long_name = "longitude of grid cell centers" ;
        geolon_t:units = "degrees_E" ;
    double time(time) ;
        time:long_name = "time" ;
        time:units = "days since 1900-01-01 00:00:00" ;
}
```

This is a database for the C48 Cubed Sphere grid defined in 12.8.5. Note currently we assume that the data are located at the center stagger of the grid. The coordinate variables `geolon_t` and `geolat_t` should be identical to the center coordinates defined in the corresponding tile files. They are not used to create the multi-tile grid. For this application, they are only used to construct the analytic field to check the correctness of the regridding results if `-check` argument is given.

If the variable specified for the destination file does not already exist in the file, the file type is determined as follows: First search for a variable that has a `cf_role` attribute of value `mesh_topology`. If successful, the file is a UGRID file. The destination variable will be created on the nodes if the regrid method is non-conservative and an optional argument `dst_loc` is set to `corner`. Otherwise, the destination variable will be created on the face. If the destination file is not a UGRID file, check if there is a variable with its `units` attribute set to `degrees_east` and another variable with it's `units` attribute set to `degrees_west`. If such a pair is found, the file is a GRIDSPEC file and the above two variables will be used as the coordinate variables for the variable to be created. If more than one pair of coordinate variables are found in the file, the application will fail with an error message.

If the destination variable exists in the destination grid file, it has to have the same number of dimensions and the same type as the source variable. Except for the latitude and longitude dimensions, the size of the destination variable's extra dimensions (e.g., time and vertical layers) has to match with the source variable. If the destination variable does not exist in the destination grid file, a new variable will be created with the same type and matching dimensions as the source variable. All the attributes of the source variable will be copied to the destination variable except those related to the grid definition (i.e. `coordinates` attribute if the destination file is in GRIDSPEC or MOSAIC format or `mesh` and `location` attributes if the destination file is in UGRID format.

Additional rules beyond the CF convention are adopted to determine whether there is a time dimension defined in the source and destination files. In this application, only a dimension with a name `time` is considered as a time dimension. If the source variable has a `time` dimension and the destination variable is not already defined, the application first checks if there is a `time` dimension defined in the destination file. If so, the values of the `time` dimension in both files have to be identical. If the time dimension values don't match, the application terminates with an error message. The application does not check the existence of a `time` variable or if the `units` attribute of the `time` variable match in two input files. If the destination file does not have a `time` dimension, it will be created. UNLIMITED time dimension is allowed in the source file, but the `time` dimension created in the destination file is not UNLIMITED.

This application requires the NetCDF library to read the grid files and write out the interpolated variables. To compile ESMF with the NetCDF library, please refer to the "Third Party Libraries" Section in the ESMF User's Guide for more information.

Internally this application uses the ESMF public API to perform regridding. If a source or destination grid is logically rectangular, then `ESMF_GridCreate()( ??)` is used to create an ESMF_Grid object from the file. The coordinate variables are stored at the center stagger location (`ESMF_STAGGERLOC_CENTER`). If the application performs a conservative regridding, the `addCornerStager` argument is set to `TRUE` and the bound variables in the grid file will be read in and stored at the corner stagger location (`ESMF_STAGGERLOC_CORNER`). If the variable has an

_FillValue attribute defined, a mask will be generated using the missing values of the variable. The data variable is defined as a ESMF_Field object at the center stagger location (ESMF_STAGGERLOC_CENTER) of the grid.

If the source grid is an unstructured grid and the the regrid method is nearest neighbor, or if the destination grid is unstructured and the regrid method is non-conservative, ESMF_LocStreamCreate()(**??** is used to create an ESMF_LocStream object. Otherwise, ESMF_MeshCreate()(**??**) is used to create an ESMF_Mesh object for the unstructured input grids. Currently, only the 2D unstructured grid is supported. If the application performs a conservative regridding, the variable has to be defined on the face of the mesh cells, i.e., its location attribute has to be set to face. Otherwise, the variable has to be defined on the node and its (location attribute is set to node).

If a source or a destination grid is a Cubed Sphere grid defined in GRIDSPEC MOSAIC file format, ESMF_GridCreateMosaic()(**??**) will be used to create a multi-tile ESMF_Grid object from the file. The coordinates at the center and the corner stagger in the tile files will be stored in the grid. The data has to be located at the center stagger of the grid.

Similar to the ESMF_RegridWeightGen command line tool (Section 12), this application supports bilinear, patch, nearest neighbor, first-order and second-order conservative interpolation. The descriptions of different interpolation methods can be found at Section 24.2 and Section 12. It also supports different pole methods for non-conservative interpolation and allows user to choose to ignore the errors when some of the destination points cannot be mapped by any source points.

If the optional argument -check is given, the interpolated fields will be checked agaist a synthetic field defined as follows:

13.2 Usage

The command line arguments are all keyword based. Both the long keyword prefixed with '--' or the one character short keyword prefixed with '-' are supported. The format to run the command line tool is as follows:

```
ESMF_Regrid
        --source|-s src_grid_filename
        --destination|-d dst_grid_filename
 --src_var var_name[,var_name,..]
 --dst_var var_name[,var_name,..]
        [--srcdatafile]
        [--dstdatafile]
        [--tilefile_path filepath]
        [--dst_loc center|corner]
        [--method|-m bilinear|patch|nearestdtos|neareststod|conserve|conserve2nd]
        [--pole|-p none|all|teeth|1|2|..]
        [--ignore_unmapped|-i]
        [--ignore_degenerate]
        [-r]
        [--src_regional]
        [--dst_regional]
        [--check]
        [--no_log]
        [--help|-h]
        [--version]
```

```
       [-V]
where
  --source or -s      - a required argument specifying the source grid
                        file name

  --destination or -d - a required argument specifying the destination
                        grid file name

  --src_var           - a required argument specifying the variable names
                        in the src grid file to be interpolated from.  If more
                        than one, separated them with comma.

  --dst_var           - a required argument specifying the variable names
                        to be interpolated to.  If more than one, separated
                        them with comma. The variable may or may not
                        exist in the destination grid file.

  --srcdatafile       - If the source grid is a GRIDSPEC MOSAIC grid, the data
                        is stored in separate files, one per tile. srcdatafile
                        is the prefix of the source data file.  The filename
                        is srcdatafile.tilename.nc, where tilename is the tile
                        name defined in the MOSAIC file.

  --dstdatafile       - If the destination grid is a GRIDSPEC MOSAIC grid, the data
                        is stored in separate files, one per tile. dstdatafile
                        is the prefix of the destination data file.  The filename
                        is dstdatafile.tilename.nc, where tilename is the tile
                        name defined in the MOSAIC file.

  --tilefile_path    - the alternative file path for the tile files and the
                        data files when either the source or the destination grid
                        is a GRIDSPEC MOSAIC grid.  The path can be either relative
                        or absolute.  If it is relative, it is relative to the
                        working directory.  When specified, the gridlocation variable
                        defined in the Mosaic file will be ignored.

    --dst_loc         - an optional argument that specifies whether the destination
                        variable is located at the center or the corner of the grid
                        if the destination variable does not exist in the destination
                        grid file. This flag is only required for non-conservative
                        regridding when the destination grid is in UGRID format.
                        For all other cases, only the center location is supported
                        that is also the default value if this argument is not specified.

  --method or -m      - an optional argument specifying which interpolation
                        method is used. The value can be one of the following:

                        bilinear   - for bilinear interpolation, also the
                                     default method if not specified.
                        patch      - for patch recovery interpolation
                        nearstdtos - for nearest destination to source interpolation
```

```
                    nearststod - for nearest source to destination interpolation
                    conserve   - for first-order conservative interpolation

--pole or -p        - an optional argument indicating what to do with
                      the pole.
                      The value can be one of the following:

                      none  - No pole, the source grid ends at the top
                              (and bottom) row of nodes specified in
                              <source grid>.
                      all   - Construct an artificial pole placed in the
                              center of the top (or bottom) row of nodes,
                              but projected onto the sphere formed by the
                              rest of the grid. The value at this pole is
                              the average of all the pole values. This
                              is the default option.

                      teeth - No new pole point is constructed, instead
                              the holes at the poles are filled by
                              constructing triangles across the top and
                              bottom row of the source Grid. This can be
                              useful because no averaging occurs, however,
                              because the top and bottom of the sphere are
                              now flat, for a big enough mismatch between
                              the size of the destination and source pole
                              regions, some destination points may still
                              not be able to be mapped to the source Grid.

                      <N>   - Construct an artificial pole placed in the
                              center of the top (or bottom) row of nodes,
                              but projected onto the sphere formed by the
                              rest of the grid. The value at this pole is
                              the average of the N source nodes next to
                              the pole and surrounding the destination
                              point (i.e.  the value may differ for each
                              destination point. Here N ranges from 1 to
                              the number of nodes around the pole.

  --ignore_unmapped
         or
         -i         - ignore unmapped destination points. If not specified
                      the default is to stop with an error if an unmapped
                      point is found.

  --ignore_degenerate - ignore degenerate cells in the input grids. If not specified
                      the default is to stop with an error if an degenerate
                      cell is found.

  -r                - an optional argument specifying that the source and
                      destination grids are regional grids.  If the argument
                      is not given, the grids are assumed to be global.
```

```
    --src_regional    - an optional argument specifying that the source is
                        a regional grid and the destination is a global grid.

    --dst_regional    - an optional argument specifying that the destination
                        is a regional grid and the source is a global grid.

    --check           - Check the correctness of the interpolated destination
                        variables against an analytic field. The source variable
                        has to be synthetically constructed using the same analytic
                        method in order to perform meaningful comparison.
                        The analytic field is calculated based on the coordinate
                        of the data point.  The formular is as follows:
                        data(i,j,k,l)=2.0+cos(lat(i,j))**2*cos(2.0*lon(i,j))+(k-1)+2*(l-1)
                        The data field can be up to four dimensional with the
                        first two dimension been longitude and latitude.
                        The mean relative error between the destination and
                        analytic field is computed.

     --no_log         - Turn off the ESMF error log.

     --help or -h     - Print the usage message and exit.

     --version        - Print ESMF version and license information and exit.

     -V               - Print ESMF version number and exit.
```

13.3 Examples

The example below regrids the node variable `zeta` defined in the sample UGRID file(13.1) to the destination grid defined in the sample GRIDSPEC file(13.1) using bilinear regridding method and write the interpolated data into a variable named `zeta`.

```
  mpirun -np 4 ESMF_Regrid -s simple_ugrid.nc -d simple_gridspec.nc \
               --src_var zeta --dst_var zeta
```

In this case, the destination variable does not exist in `simple_ugrid.nc` and the `time` dimension is not defined in the destination file. The resulting output file has a new time dimension and a new variable `zeta`. The attributes from the source variable `zeta` are copied to the destination variable except for `mesh` and `location`. A new attribute `coordinates` is created for the destination variable to specify the names of the coordinate variables. The header of the output file looks like:

```
netcdf simple_gridspec {
dimensions:
      lat = 192 ;
      lon = 288 ;
      time = 2  ;
```

```
variables:
      float PSL(lat, lon) ;
         PSL:time = 50. ;
         PSL:units = "Pa" ;
         PSL:long_name = "Sea level pressure" ;
         PSL:cell_method = "time: mean" ;
         PSL:coordinates = "lon lat" ;
      double lat(lat) ;
         lat:long_name = "latitude" ;
         lat:units = "degrees_north" ;
      double lon(lon) ;
         lon:long_name = "longitude" ;
         lon:units = "degrees_east" ;
      float zeta(time, lat, lon) ;
         zeta:standard_name = "sea_surface_height_above_geoid" ;
         zeta:_FillValue = -999. ;
         zeta:coordinates = "lon lat" ;
}
```

The next example shows the command to do the same thing as the previous example but for a different variable `ua`. Since `ua` is defined on the face, we can only do a conservative regridding.

```
  mpirun -np 4 ESMF_Regrid -s simple_ugrid.nc -d simple_gridspec.nc \
               --src_var ua --dst_var ua -m conserve
```

14 ESMF_Scrip2Unstruct

14.1 Description

The `ESMF_Scrip2Unstruct` application is a parallel program that converts a SCRIP format grid file 12.8.1 into an unstructured grid file in the ESMF unstructured file format 12.8.2 or in the UGRID file format 12.8.4. This application program can be used together with `ESMF_RegridWeightGen` 12 application for the unstructured SCRIP format grid files. An unstructured SCRIP grid file will be converted into the ESMF unstructured file format internally in `ESMF_RegridWeightGen`. The conversion subroutine used in `ESMF_RegridWeightGen` is sequential and could be slow if the grid file is very big. It will be more efficient to run the `ESMF_Scrip2Unstruct` first and then regrid the output ESMF or UGRID file using `ESMF_RegridWeightGen`. Note that a logically rectangular grid file in the SCRIP format (i.e. the dimension `grid_rank` is equal to 2) can also be converted into an unstructured grid file with this application.

The application usage is as follows:

```
ESMF_Scrip2Unstruct  inputfile outputfile dualflag [fileformat]

where
  inputfile       - a SCRIP format grid file

  outputfile      - the output file name
```

```
dualflag        - 0 for straight conversion and 1 for dual
    mesh.  A dual mesh is a mesh constructed
                  by putting the corner coordinates in the
                  center of the elements and using the
    center coordinates to form the mesh
    corner vertices.

fileformat      - an optional argument for the output file
    format.  It could be either ESMF or UGRID.
                  If not specified, the output file is in
    the ESMF format.
```

Part III

Superstructure

15 Overview of Superstructure

ESMF superstructure classes define an architecture for assembling Earth system applications from modeling **components**. A component may be defined in terms of the physical domain that it represents, such as an atmosphere or sea ice model. It may also be defined in terms of a computational function, such as a data assimilation system. Earth system research often requires that such components be **coupled** together to create an application. By coupling we mean the data transformations and, on parallel computing systems, data transfers, that are necessary to allow data from one component to be utilized by another. ESMF offers regridding methods and other tools to simplify the organization and execution of inter-component data exchanges.

In addition to components defined at the level of major physical domains and computational functions, components may be defined that represent smaller computational functions within larger components, such as the transformation of data between the physics and dynamics in a spectral atmosphere model, or the creation of nested higher resolution regions within a coarser grid. The objective is to couple components at varying scales both flexibly and efficiently. ESMF encourages a hierarchical application structure, in which large components branch into smaller sub-components (see Figure 2). ESMF also makes it easier for the same component to be used in multiple contexts without changes to its source code.

Key Features
Modular, component-based architecture.
Hierarchical assembly of components into applications.
Use of components in multiple contexts without modification.
Sequential or concurrent component execution.
Single program, multiple datastream (SPMD) applications for maximum portability and reconfigurability.
Multiple program, multiple datastream (MPMD) option for flexibility.

15.1 Superstructure Classes

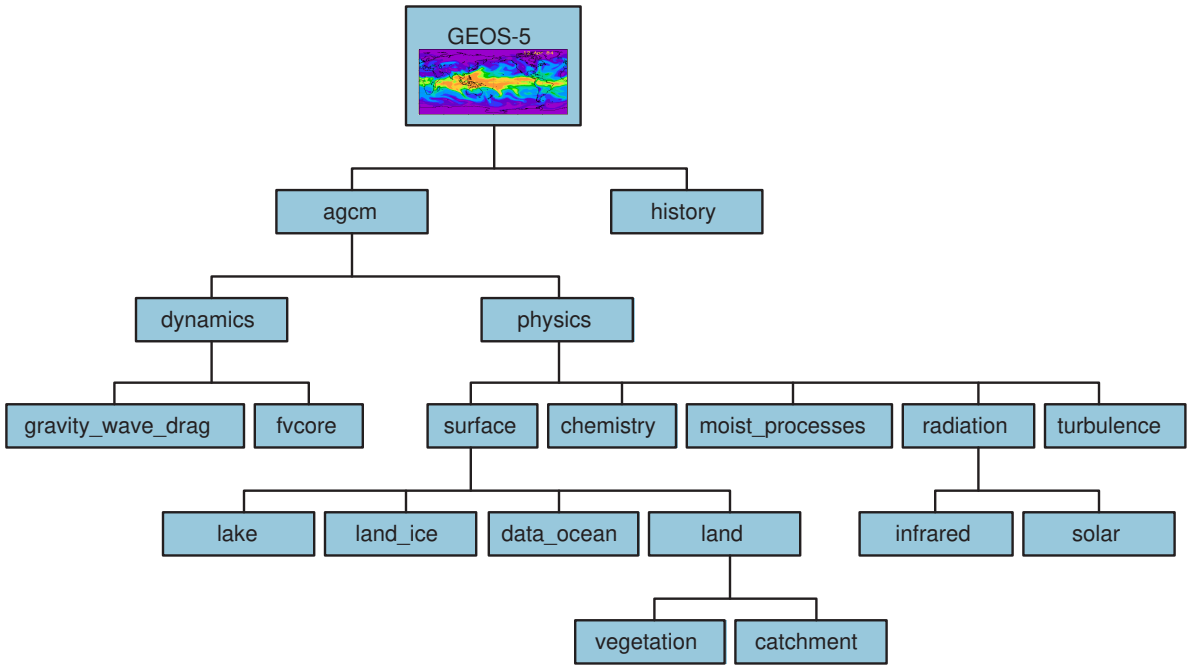
There are a small number of classes in the ESMF superstructure:

- **Component** An ESMF component has two parts, one that is supplied by ESMF and one that is supplied by the user. The part that is supplied by the framework is an ESMF derived type that is either a Gridded Component (**GridComp**) or a Coupler Component (**CplComp**). A Gridded Component typically represents a physical domain in which data is associated with one or more grids - for example, a sea ice model. A Coupler Component arranges and executes data transformations and transfers between one or more Gridded Components. Gridded Components and Coupler Components have standard methods, which include initialize, run, and finalize. These methods can be multi-phase.

  The second part of an ESMF Component is user code, such as a model or data assimilation system. Users set entry points within their code so that it is callable by the framework. In practice, setting entry points means that within user code there are calls to ESMF methods that associate the name of a Fortran subroutine with a corresponding standard ESMF operation. For example, a user-written initialization routine called `myOceanInit` might be associated with the standard initialize routine of an ESMF Gridded Component named "myOcean" that represents an ocean model.

- **State** ESMF Components exchange information with other Components only through States. A State is an ESMF derived type that can contain Fields, FieldBundles, Arrays, ArrayBundles, and other States. A Component is associated with two States, an **Import State** and an **Export State**. Its Import State holds the data that it receives from other Components. Its Export State contains data that it makes available to other Components.

Figure 2: ESMF enables applications such as the atmospheric general circulation model GEOS-5 to be structured hierarchically, and reconfigured and extended easily. Each box in this diagram is an ESMF Gridded Component.



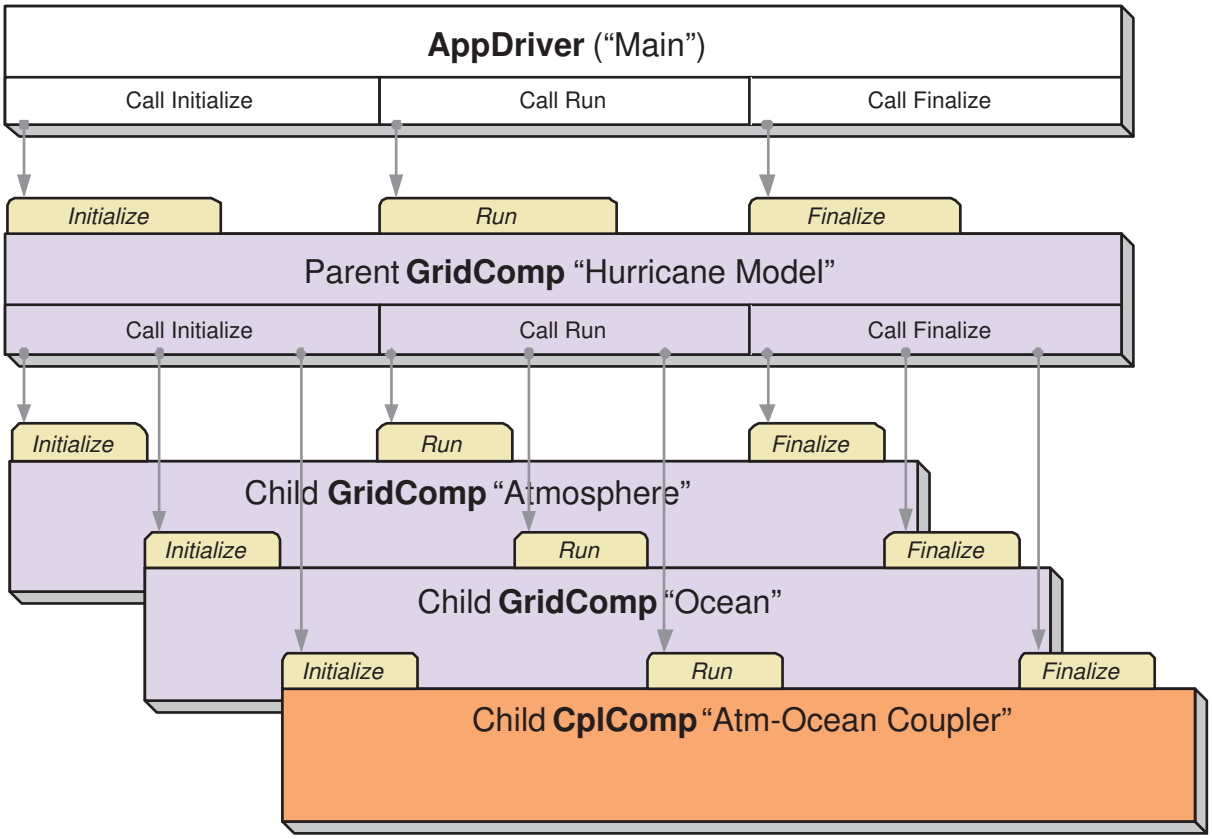
An ESMF coupled application typically involves a parent Gridded Component, two or more child Gridded Components and one or more Coupler Components.

The parent Gridded Component is responsible for creating the child Gridded Components that are exchanging data, for creating the Coupler, for creating the necessary Import and Export States, and for setting up the desired sequencing. The application's "main" routine calls the parent Gridded Component's initialize, run, and finalize methods in order to execute the application. For each of these standard methods, the parent Gridded Component in turn calls the corresponding methods in the child Gridded Components and the Coupler Component. For example, consider a simple coupled ocean/atmosphere simulation. When the initialize method of the parent Gridded Component is called by the application, it in turn calls the initialize methods of its child atmosphere and ocean Gridded Components, and the initialize method of an ocean-to-atmosphere Coupler Component. Figure 3 shows this schematically.

15.2 Hierarchical Creation of Components

Components are allocated computational resources in the form of **Persistent Execution Threads**, or **PET**s. A list of a Component's PETs is contained in a structure called a **Virtual Machine**, or **VM**. The VM also contains information about the topology and characteristics of the underlying computer. Components are created hierarchically, with parent Components creating child Components and allocating some or all of their PETs to each one. By default ESMF creates a new VM for each child Component, which allows Components to tailor their VM resources to match their needs. In some cases, a child may want to share its parent's VM - ESMF supports this, too.

Figure 3: A call to a standard ESMF initialize (run, finalize) method by a parent component triggers calls to initialize (run, finalize) all of its child components.



A Gridded Component may exist across all the PETs in an application. A Gridded Component may also reside on a subset of PETs in an application. These PETs may wholly coincide with, be wholly contained within, or wholly contain another Component.

15.3 Sequential and Concurrent Execution of Components

When a set of Gridded Components and a Coupler runs in sequence on the same set of PETs the application is executing in a **sequential** mode. When Gridded Components are created and run on mutually exclusive sets of PETs, and are coupled by a Coupler Component that extends over the union of these sets, the mode of execution is **concurrent**.

Figure 4 illustrates a typical configuration for a simple coupled sequential application, and Figure 5 shows a possible configuration for the same application running in a concurrent mode.

Parent Components can select if and when to wait for concurrently executing child Components, synchronizing only when required.

It is possible for ESMF applications to contain some Component sets that are executing sequentially and others that are executing concurrently. We might have, for example, atmosphere and land Components created on the same subset of PETs, ocean and sea ice Components created on the remainder of PETs, and a Coupler created across all the PETs in the application.

15.4 Intra-Component Communication

All data transfers within an ESMF application occur *within* a component. For example, a Gridded Component may contain halo updates. Another example is that a Coupler Component may redistribute data between two Gridded Components. As a result, the architecture of ESMF does not depend on any particular data communication mechanism, and new communication schemes can be introduced without affecting the overall structure of the application.

Since all data communication happens within a component, a Coupler Component must be created on the union of the PETs of all the Gridded Components that it couples.

15.5 Data Distribution and Scoping in Components

The scope of distributed objects is the VM of the currently executing Component. For this reason, all PETs in the current VM must make the same distributed object creation calls. When a Coupler Component running on a superset of a Gridded Component's PETs needs to make communication calls involving objects created by the Gridded Component, an ESMF-supplied function called `ESMF_StateReconcile()` creates proxy objects for those PETs that had no previous information about the distributed objects. Proxy objects contain no local data but can be used in communication calls (such as regrid or redistribute) to describe the remote source for data being moved to the current PET, or to describe the remote destination for data being moved from the local PET. Figure 6 is a simple schematic that shows the sequence of events in a reconcile call.

15.6 Performance

The ESMF design enables the user to configure ESMF applications so that data is transferred directly from one component to another, without requiring that it be copied or sent to a different data buffer as an interim step. This is likely to be the most efficient way of performing inter-component coupling. However, if desired, an application can also be configured so that data from a source component is sent to a distinct set of Coupler Component PETs for processing before being sent to its destination.

The ability to overlap computation with communication is essential for performance. When running with ESMF the user can initiate data sends during Gridded Component execution, as soon as the data is ready. Computations can then proceed simultaneously with the data transfer.

Figure 4: Schematic of the run method of a coupled application, with an “Atmosphere” and an “Ocean” Gridded Component running sequentially with an “Atm-Ocean Coupler.” The top-level “Hurricane Model” Gridded Component contains the sequencing information and time advancement loop. The application driver, Coupler, and all Gridded Components are distributed over nine PETs.

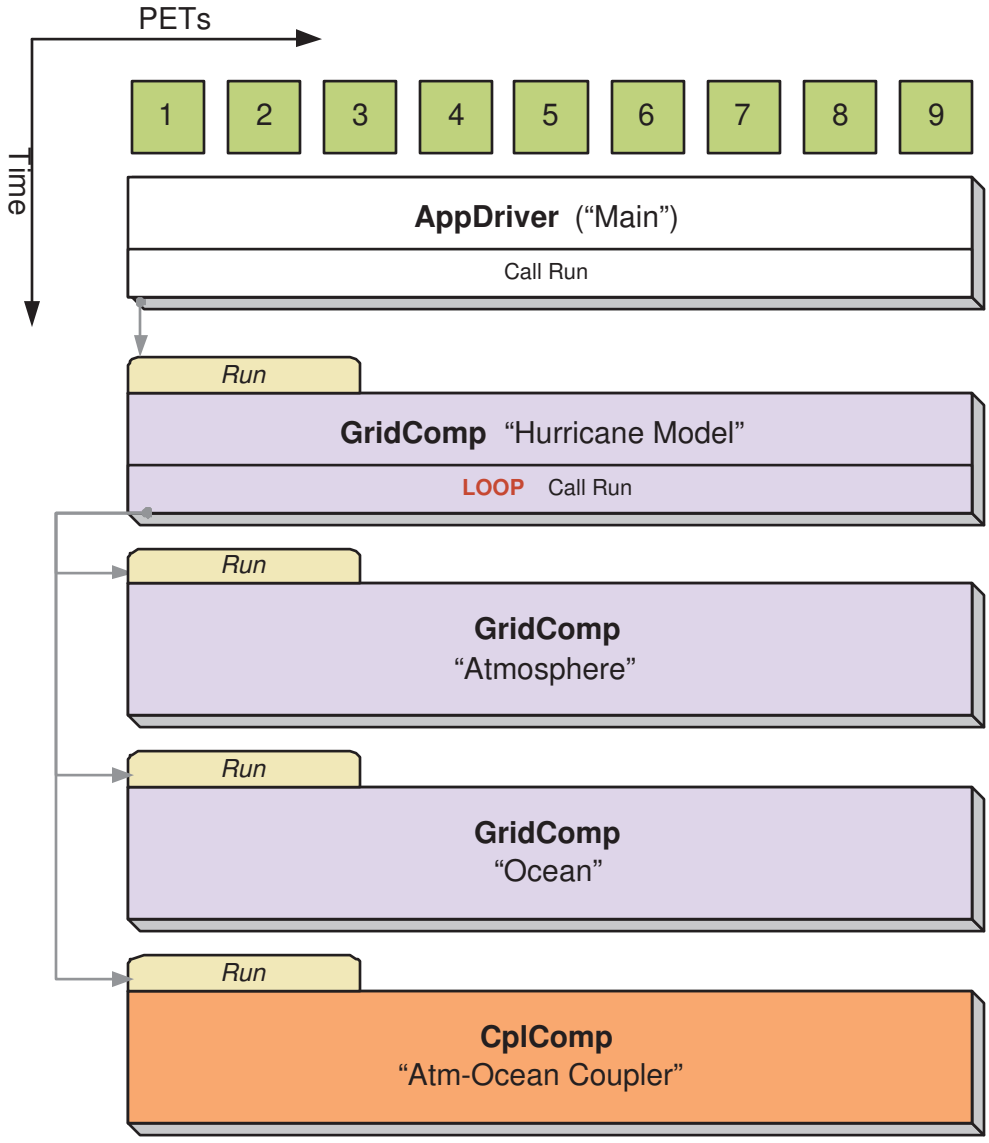


Figure 5: Schematic of the run method of a coupled application, with an “Atmosphere” and an “Ocean” Gridded Component running concurrently with an “Atm-Ocean Coupler.” The top-level “Hurricane Model” Gridded Component contains the sequencing information and time advancement loop. The application driver, Coupler, and top-level “Hurricane Model” Gridded Component are distributed over nine PETs. The “Atmosphere” Gridded Component is distributed over three PETs and the “Ocean” Gridded Component is distributed over six PETs.

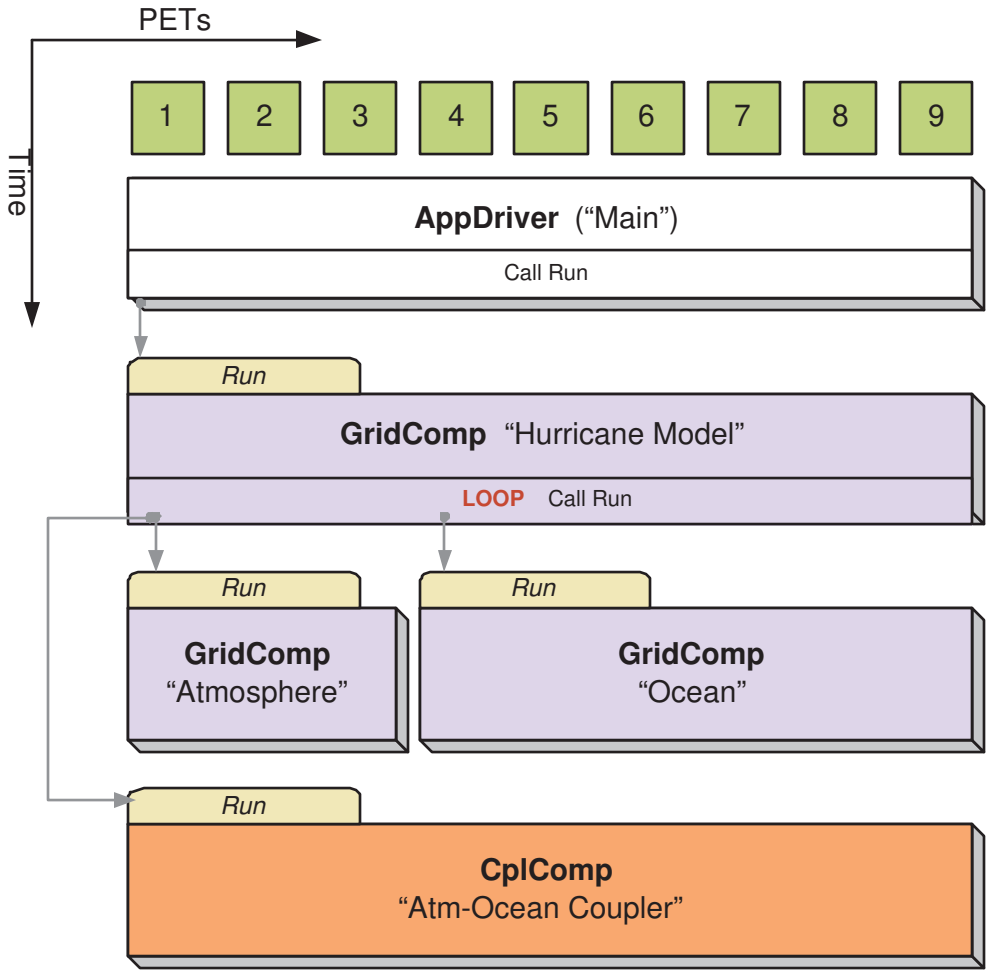
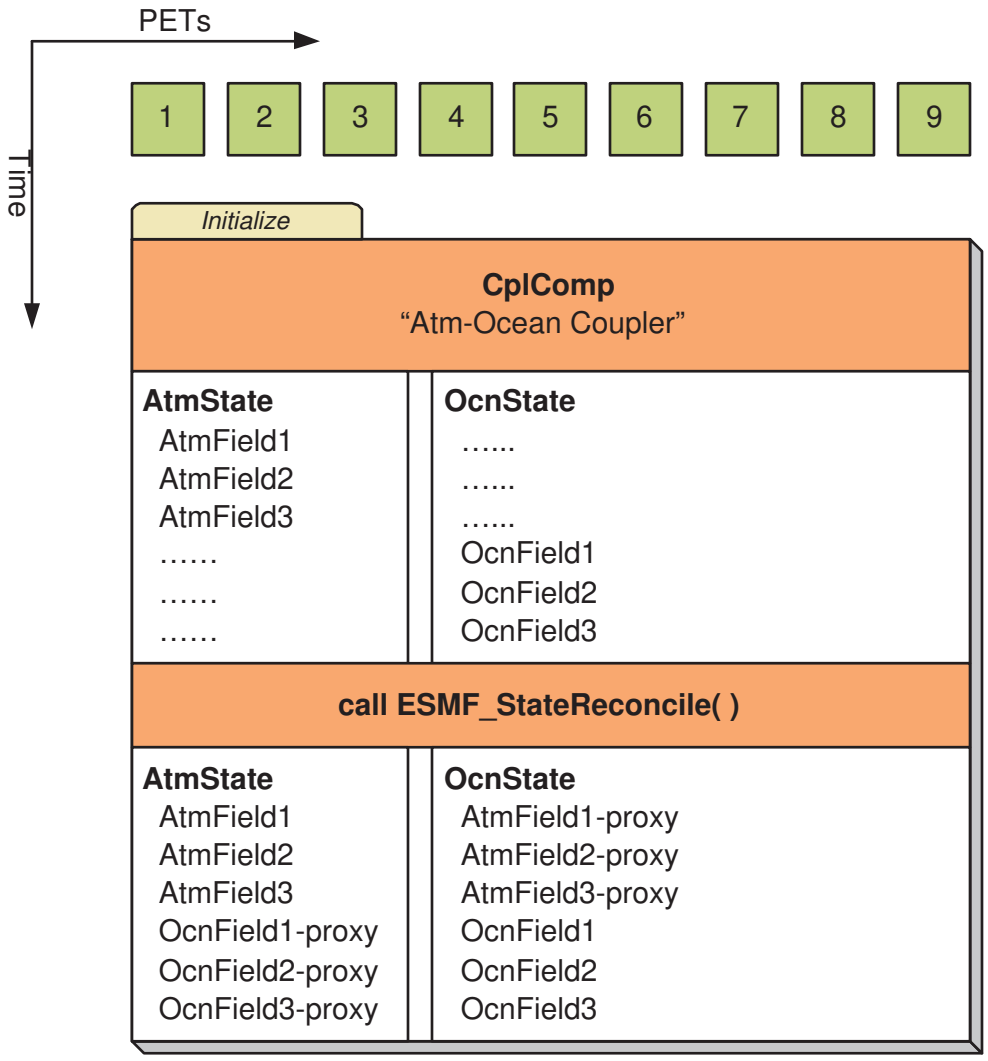
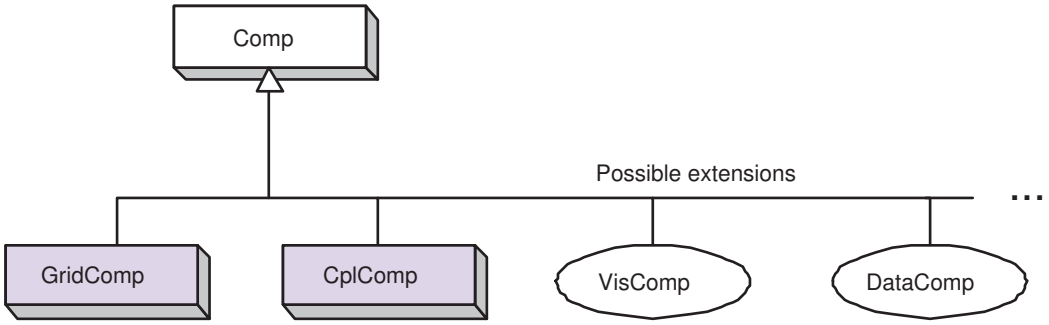


Figure 6: An `ESMF_StateReconcile()` call creates proxy objects for use in subsequent communication calls. The reconcile call would normally be made during Coupler initialization.



15.7 Object Model

The following is a simplified Unified Modeling Language (UML) diagram showing the relationships among ESMF superstructure classes. See Appendix A, *A Brief Introduction to UML*, for a translation table that lists the symbols in the diagram and their meaning.



16 Application Driver and Required ESMF Methods

16.1 Description

Every ESMF application needs a driver code. Typically the driver layer is implemented as the "main" of the application, although this is not strictly an ESMF requirement. For most ESMF applications the task of the application driver will be very generic: Initialize ESMF, create a top-level Component and call its Initialize, Run and Finalize methods, before destroying the top-level Component again and calling ESMF Finalize.

ESMF provides a number of different application driver templates in the `$ESMF_DIR/src/Superstructure/AppDriver` directory. An appropriate one can be chosen depending on how the application is to be structured:

Sequential vs. Concurrent Execution In a sequential execution model, every Component executes on all PETs, with each Component completing execution before the next Component begins. This has the appeal of simplicity of data consumption and production: when a Gridded Component starts, all required data is available for use, and when a Gridded Component finishes, all data produced is ready for consumption by the next Gridded Component. This approach also has the possibility of less data movement if the grid and data decomposition is done such that each processor's memory contains the data needed by the next Component.

In a concurrent execution model, subgroups of PETs run Gridded Components and multiple Gridded Components are active at the same time. Data exchange must be coordinated between Gridded Components so that data deadlock does not occur. This strategy has the advantage of allowing coupling to other Gridded Components at any time during the computational process, including not having to return to the calling level of code before making data available.

Pairwise vs. Hub and Spoke Coupler Components are responsible for taking data from one Gridded Component and putting it into the form expected by another Gridded Component. This might include regridding, change of units, averaging, or binning.

Coupler Components can be written for *pairwise* data exchange: the Coupler Component takes data from a single Component and transforms it for use by another single Gridded Component. This simplifies the structure of the Coupler Component code.

Couplers can also be written using a *hub and spoke* model where a single Coupler accepts data from all other Components, can do data merging or splitting, and formats data for all other Components.

Multiple Couplers, using either of the above two models or some mixture of these approaches, are also possible.

Implementation Language The ESMF framework currently has Fortran interfaces for all public functions. Some functions also have C interfaces, and the number of these is expected to increase over time.

Number of Executables The simplest way to run an application is to run the same executable program on all PETs. Different Components can still be run on mutually exclusive PETs by using branching (e.g., if this is PET 1, 2, or 3, run Component A, if it is PET 4, 5, or 6 run Component B). This is a **SPMD** model, Single Program Multiple Data.

The alternative is to start a different executable program on different PETs. This is a **MPMD** model, Multiple Program Multiple Data. There are complications with many job control systems on multiprocessor machines in getting the different executables started, and getting inter-process communications established. ESMF currently has some support for MPMD: different Components can run as separate executables, but the Coupler that transfers data between the Components must still run on the union of their PETs. This means that the Coupler Component must be linked into all of the executables.

16.2 Constants

16.2.1 ESMF_END

DESCRIPTION:

The `ESMF_End_Flag` determines how an ESMF application is shut down.

The type of this flag is:

```
type(ESMF_End_Flag)
```

The valid values are:

ESMF_END_ABORT Global abort of the ESMF application. There is no guarantee that all PETs will shut down cleanly during an abort. However, all attempts are made to prevent the application from hanging and the LogErr of at least one PET will be completely flushed during the abort. This option should only be used if a condition is detected that prevents normal continuation or termination of the application. Typical conditions that warrant the use of `ESMF_END_ABORT` are those that occur on a per PET basis where other PETs may be blocked in communication calls, unable to reach the normal termination point. An aborted application returns to the parent process with a system dependent indication that a failure occurred during execution.

ESMF_END_NORMAL Normal termination of the ESMF application. Wait for all PETs of the global VM to reach `ESMF_Finalize()` before termination. This is the clean way of terminating an application. `MPI_Finalize()` will be called in case of MPI applications.

ESMF_END_KEEPMPI Same as `ESMF_END_NORMAL` but `MPI_Finalize()` will *not* be called. It is the user code's responsibility to shut down MPI cleanly if necessary.

16.3 Use and Examples

ESMF encourages application organization in which there is a single top-level Gridded Component. This provides a simple, clear sequence of operations at the highest level, and also enables the entire application to be treated as a sub-Component of another, larger application if desired. When a simple application is organized in this fashion the standard AppDriver can probably be used without much modification.

Examples of program organization using the AppDriver can be found in the `src/Superstructure/AppDriver` directory. A set of subdirectories within the AppDriver directory follows the naming convention:

```
<seq|concur>_<pairwise|hub>_<f|c>driver_<spmd|mpmd>
```

The example that is currently implemented is `seq_pairwise_fdriver_spmd`, which has sequential component execution, a pairwise coupler, a main program in Fortran, and all processors launching the same executable. It is also copied automatically into a top-level `quick_start` directory at compilation time.

The user can copy the AppDriver files into their own local directory. Some of the files can be used unchanged. Others are template files which have the rough outline of the code but need additional application-specific code added in order to perform a meaningful function. The `README` file in the AppDriver subdirectory or `quick_start` directory contains instructions about which files to change.

Examples of concurrent component execution can be found in the system tests that are bundled with the ESMF distribution.

16.4 Required ESMF Methods

There are a few methods that every ESMF application must contain. First, `ESMF_Initialize()` and `ESMF_Finalize()` are in complete analogy to `MPI_Init()` and `MPI_Finalize()` known from MPI. All ESMF programs, serial or parallel, must initialize the ESMF system at the beginning, and finalize it at the end of execution. The behavior of calling any ESMF method before `ESMF_Initialize()`, or after `ESMF_Finalize()` is undefined.

Second, every ESMF Component that is accessed by an ESMF application requires that its set services routine is called through `ESMF_<Grid/Cpl>CompSetServices()`. The Component must implement one public entry point, its set services routine, that can be called through the `ESMF_<Grid/Cpl>CompSetServices()` library routine. The Component set services routine is responsible for setting entry points for the standard ESMF Component methods Initialize, Run, and Finalize.

Finally, the Component can optionally call `ESMF_<Grid/Cpl>CompSetVM()` *before* calling `ESMF_<Grid/Cpl>CompSetServices()`. Similar to `ESMF_<Grid/Cpl>CompSetServices()`, the `ESMF_<Grid/Cpl>CompSetVM()` call requires a public entry point into the Component. It allows the Component to adjust certain aspects of its execution environment, i.e. its own VM, before it is started up.

The following sections discuss the above mentioned aspects in more detail.

16.4.1 User-code `SetServices` method

Many programs call some library routines. The library documentation must explain what the routine name is, what arguments are required and what are optional, and what the code does.

In contrast, all ESMF components must be written to *be called* by another part of the program; in effect, an ESMF component takes the place of a library. The interface is prescribed by the framework, and the component writer must

provide specific subroutines which have standard argument lists and perform specific operations. For technical reasons *none* of the arguments in user-provided subroutines must be declared as *optional*.

The only *required* public interface of a Component is its SetServices method. This subroutine must have an externally accessible name (be a public symbol), take a component as the first argument, and an integer return code as the second. Both arguments are required and must *not* be declared as `optional`. If an intent is specified in the interface it must be `intent(inout)` for the first and `intent(out)` for the second argument. The subroutine name is not predefined, it is set by the component writer, but must be provided as part of the component documentation.

The required function that the SetServices subroutine must provide is to specify the user-code entry points for the standard ESMF Component methods. To this end the user-written SetServices routine calls the

`ESMF_<Grid/Cpl>CompSetEntryPoint()` method to set each Component entry point.

See sections **??** and **??** for examples of how to write a user-code SetServices routine.

Note that a component does not call its own SetServices routine; the AppDriver or parent component code, which is creating a component, will first call `ESMF_<Grid/Cpl>CompCreate()` to create a component object, and then must call into `ESMF_<Grid/Cpl>CompSetServices()`, supplying the user-code SetServices routine as an argument. The framework then calls into the user-code SetServices, after the Component's VM has been started up.

It is good practice to package the user-code implementing a component into a Fortran module, with the user-code SetService routine being the only public module method. ESMF supports three mechanisms for accessing the user-code SetServices routine from the calling AppDriver or parent component.

- **Fortran USE association**: The AppDriver or parent component utilizes the standard Fortran USE statement on the component module to make all public entities available. The user-code SetServices routine can then be passed directly into the `ESMF_<Grid/Cpl>CompSetServices()` interface documented in **??** and **??**, respectively.

  Pros: Standard Fortran module use: name mangling and interface checking is handled by the Fortran compiler.

  Cons: Fortran 90/95 has no mechanism to implement a "smart" dependency scheme through USE association. Any change in a lower level component module (even just adding or changing a comment!) will trigger a complete recompilation of all of the higher level components throughout the component hierarchy. This situation is particularly annoying for ESMF componentized code, where the prescribed ESMF component interfaces, in principle, remove all interdependencies between components that would require recompilation.

  Fortran *submodules*, introduced as an extension to Fortran 2003, and now part for the Fortran 2008 standard, are designed to avoid this "false" dependency issue. A code change to an ESMF component that keeps the actual implementation within a submodule, will not trigger a recompilation of the components further up in the component hierarchy. Unfortunately, as of mid-2015, only two compiler vendors support submodules.

- **External routine**: The AppDriver or parent component provides an explicit interface block for an external routine that implements (or calls) the user-code SetServices routine. This routine can then be passed directly into the `ESMF_<Grid/Cpl>CompSetServices()` interface documented in **??** and **??**, respectively. (In practice this can be implemented by the component as an external subroutine that simply calls into the user-code SetServices module routine.)

  Pros: Avoids Fortran USE dependencies: a change to lower level component code will not trigger a complete recompilation of all of the higher level components throughout the component hierarchy. Name mangling is handled by the Fortran compiler.

  Cons: The user-code SetServices interface is not checked by the compiler. The user must ensure uniqueness of the external routine name across the entire application.

- **Name lookup**: The AppDriver or parent component specifies the user-code SetServices routine by name. The actual lookup and code association does not occur until runtime. The name string is passed into the

ESMF_<Grid/Cpl>CompSetServices() interface documented in **??** and **??**, respectively.

Pros: Avoids Fortran USE dependencies: a change to lower level component code will not trigger a complete recompilation of all of the higher level components throughout the component hierarchy. The component code does not have to be accessible until runtime and may be located in a shared object, thus avoiding relinking of the application.

Cons: The user-code SetServices interface is not checked by the compiler. The user must explicitly deal with all of the Fortran name mangling issues: 1) Accessing a module routine requires precise knowledge of the name mangling rules of the specific compiler. Alternatively, the user-code SetServices routine may be implemented as an external routine, avoiding the module name mangling. 2) Even then, Fortran compilers typically append one or two underscores on a symbol name. This must be considered when passing the name into the ESMF_<Grid/Cpl>CompSetServices() method.

16.4.2 User-code Initialize, Run, and Finalize methods

The required standard ESMF Component methods, for which user-code entry points must be set, are Initialize, Run, and Finalize. Currently optional, a Component may also set entry points for the WriteRestart and ReadRestart methods.

Sections **??** and **??** provide examples of how the entry points for Initialize, Run, and Finalize are set during the user-code SetServices routine, using the ESMF_<Grid/Cpl>CompSetEntryPoint() library call.

All standard user-code methods must abide *exactly* to the prescribed interfaces. *None* of the arguments must be declared as *optional*.

The names of the Initialize, Run, and Finalize user-code subroutines do not need to be public; in fact it is far better for them to be private to lower the chances of public symbol clashes between different components.

See sections **??**, **??**, **??**, and **??**, **??**, **??** for examples of how to write entry points for the standard ESMF Component methods.

16.4.3 User-code SetVM method

When the AppDriver or parent component code calls ESMF_<Grid/Cpl>CompCreate() it has the option to specify a petList argument. All of the parent PETs contained in this list become resources of the child component. By default, without the petList argument, all of the parent PETs are provided to the child component.

Typically each component has its own virtual machine (VM) object. However, using the optional contextflag argument during ESMF_<Grid/Cpl>CompCreate() a child component can inherit its parent component's VM. Unless a child component inherits the parent VM, it has the option to set certain aspects of how its VM utilizes the provided resources. The resources provided via the parent PETs are the associated processing elements (PEs) and virtual address spaces (VASs).

The optional user-written SetVM routine is called from the parent for the child through the ESMF_<Grid/Cpl>CompSetVM() method. This is the only place where the child component can set aspects of its own VM before it is started up. The child component's VM must be running before the SetServices routine can be called, and thus the parent must call the optional ESMF_<Grid/Cpl>CompSetVM() method *before* ESMF_<Grid/Cpl>CompSetServices().

Inside the user-code called by the SetVM routine, the component has the option to specify how the PETs share the provided parent PEs. Further, PETs on the same single system image (SSI) can be set to run multi-threaded within a reduced number of virtual address spaces (VAS), allowing a component to leverage shared memory concepts.

Sections **??** and **??** provide examples for simple user-written SetVM routines.

One common use of the SetVM approach is to implement hybrid parallelism based on MPI+OpenMP. Under ESMF, each component can use its own hybrid parallelism implementation. Different components, even if running on the same PE resources, do not have to agree on the number of MPI processes (i.e. PETs), or the number of OpenMP threads launched under each PET. Hybrid and non-hybrid components can be mixed within the same application. Coupling between components of any flavor is supported under ESMF.

In order to obtain best performance when using SetVM based resource control for hybrid parallelism, it is *strongly recommended* to set `OMP_WAIT_POLICY=PASSIVE` in the environment. This is one of the standard OpenMP environment variables. The `PASSIVE` setting ensures that OpenMP threads relinquish the PEs as soon as they have completed their work. Without that setting ESMF resource control threads can be delayed, and context switching between components becomes more expensive.

16.4.4 Use of `internal procedures` as user-provided procedures

Internal procedures are nested within a surrounding procedure, and only local to the surrounding procedure. They are specified by using the CONTAINS statement.

Prior to Fortran-2008 an internal procedure could not be used as a user-provided callback procedure. In Fortran-2008 this restriction was lifted. It is important to note that if ESMF is passed an internal procedure, that the surrounding procedure be active whenever ESMF calls it. This helps ensure that local variables at the surrounding procedures scope are properly initialized.

When internal procedures contained within a main program unit are used for callbacks, there is no problem. This is because the main program unit is always active. However when internal procedures are used within other program units, initialization could become a problem. The following outlines the issue:

```
module my_procs_mod
  use ESMF
  implicit none

contains

  subroutine my_procs (...)
    integer :: my_setting
    :
    call ESMF_GridCompSetEntryPoint(gridcomp, methodflag=ESMF_METHOD_INITIALIZE, &
        userRoutine=my_grid_proc_init, rc=localrc)
    :
    my_setting = 42

  contains

    subroutine my_grid_proc_init (gridcomp, importState, exportState, clock, rc)
      :
    ! my_setting is possibly uninitialized when my_grid_proc_init is used as a call-back
      something = my_setting
      :
    end subroutine my_grid_proc_init
  end subroutine my_procs
end module my_procs_mod
```

The Fortran standard does not specify whether variable *my_setting* is statically or automatically allocated, unless it is explicitly given the SAVE attribute. Thus there is no guarantee that its value will persist after *my_procs* has finished. The SAVE attribute is usually given to a variable via specifying a SAVE attribute in its delaration. However it can also be inferred by initializing the variable in its declaration:

```
:
integer, save : my_setting
:
```

or,

```
:
integer :: my_setting = 42
:
```

Because of the potential initialization issues, it is recommended that internal procedures only be used as ESMF callbacks when the surrounding procedure is also active.

17 GridComp Class

17.1 Description

In Earth system modeling, the most natural way to think about an ESMF Gridded Component, or `ESMF_GridComp`, is as a piece of code representing a particular physical domain, such as an atmospheric model or an ocean model. Gridded Components may also represent individual processes, such as radiation or chemistry. It's up to the application writer to decide how deeply to "componentize."

Earth system software components tend to share a number of basic features. Most ingest and produce a variety of physical fields, refer to a (possibly noncontiguous) spatial region and a grid that is partitioned across a set of computational resources, and require a clock for things like stepping a governing set of PDEs forward in time. Most can also be divided into distinct initialize, run, and finalize computational phases. These common characteristics are used within ESMF to define a Gridded Component data structure that is tailored for Earth system modeling and yet is still flexible enough to represent a variety of domains.

A well designed Gridded Component does not store information internally about how it couples to other Gridded Components. That allows it to be used in different contexts without changes to source code. The idea here is to avoid situations in which slightly different versions of the same model source are maintained for use in different contexts - standalone vs. coupled versions, for example. Data is passed in and out of Gridded Components using an ESMF State, this is described in Section 21.1.

An ESMF Gridded Component has two parts, one which is user-written and another which is part of the framework. The user-written part is software that represents a physical domain or performs some other computational function. It forms the body of the Gridded Component. It may be a piece of legacy code, or it may be developed expressly for use with ESMF. It must contain routines with standard ESMF interfaces that can be called to initialize, run, and finalize the Gridded Component. These routines can have separate callable phases, such as distinct first and second initialization steps.

ESMF provides the Gridded Component derived type, `ESMF_GridComp`. An `ESMF_GridComp` must be created for every portion of the application that will be represented as a separate component. For example, in a climate model,

there may be Gridded Components representing the land, ocean, sea ice, and atmosphere. If the application contains an ensemble of identical Gridded Components, every one has its own associated ESMF_GridComp. Each Gridded Component has its own name and is allocated a set of computational resources, in the form of an ESMF Virtual Machine, or VM.

The user-written part of a Gridded Component is associated with an ESMF_GridComp derived type through a routine called ESMF_SetServices(). This is a routine that the user must write, and declare public. Inside the SetServices routine the user must call ESMF_SetEntryPoint() methods that associate a standard ESMF operation with the name of the corresponding Fortran subroutine in their user code.

17.2 Use and Examples

A Gridded Component is a computational entity which consumes and produces data. It uses a State object to exchange data between itself and other Components. It uses a Clock object to manage time, and a VM to describe its own and its child components' computational resources.

This section shows how to create Gridded Components. For demonstrations of the use of Gridded Components, see the system tests that are bundled with the ESMF software distribution. These can be found in the directory esmf/src/system_tests.

17.3 Restrictions and Future Work

1. **No optional arguments.** User-written routines called by SetServices, and registered for Initialize, Run and Finalize, *must not* declare any of the arguments as optional.

2. **Namespace isolation.** If possible, Gridded Components should attempt to make all data private, so public names do not interfere with data in other components.

3. **Single execution mode.** It is not expected that a single Gridded Component be able to function in both sequential and concurrent modes, although Gridded Components of different types can be nested. For example, a concurrently called Gridded Component can contain several nested sequential Gridded Components.

17.4 Class API

18 CplComp Class

18.1 Description

In a large, multi-component application such as a weather forecasting or climate prediction system running within ESMF, physical domains and major system functions are represented as Gridded Components (see Section 17.1). A Coupler Component, or ESMF_CplComp, arranges and executes the data transformations between the Gridded Components. Ideally, Coupler Components should contain all the information about inter-component communication for an application. This enables the Gridded Components in the application to be used in multiple contexts; that is, used in different coupled configurations without changes to their source code. For example, the same atmosphere might in one case be coupled to an ocean in a hurricane prediction model, and to a data assimilation system for numerical weather prediction in another. A single Coupler Component can couple two or more Gridded Components.

Like Gridded Components, Coupler Components have two parts, one that is provided by the user and another that is part of the framework. The user-written portion of the software is the coupling code necessary for a particular exchange

between Gridded Components. This portion of the Coupler Component code must be divided into separately callable initialize, run, and finalize methods. The interfaces for these methods are prescribed by ESMF.

The term "user-written" is somewhat misleading here, since within a Coupler Component the user can leverage ESMF infrastructure software for regridding, redistribution, lower-level communications, calendar management, and other functions. However, ESMF is unlikely to offer all the software necessary to customize a data transfer between Gridded Components. For instance, ESMF does not currently offer tools for unit tranformations or time averaging operations, so users must manage those operations themselves.

The second part of a Coupler Component is the `ESMF_CplComp` derived type within ESMF. The user must create one of these types to represent a specific coupling function, such as the regular transfer of data between a data assimilation system and an atmospheric model. [2]

The user-written part of a Coupler Component is associated with an `ESMF_CplComp` derived type through a routine called `ESMF_SetServices()`. This is a routine that the user must write and declare public. Inside the `ESMF_SetServices()` routine the user must call `ESMF_SetEntryPoint()` methods that associate a standard ESMF operation with the name of the corresponding Fortran subroutine in their user code. For example, a user routine called "couplerInit" might be associated with the standard initialize routine in a Coupler Component.

18.2 Use and Examples

A Coupler Component manages the transformation of data between Components. It contains a list of State objects and the operations needed to make them compatible, including such things as regridding and unit conversion. Coupler Components are user-written, following prescribed ESMF interfaces and, wherever desired, using ESMF infrastructure tools.

18.3 Restrictions and Future Work

1. **No optional arguments.** User-written routines called by SetServices, and registered for Initialize, Run and Finalize, *must not* declare any of the arguments as optional.

2. **No Transforms.** Components must exchange data through `ESMF_State` objects. The input data are available at the time the component code is called, and data to be returned to another component are available when that code returns.

3. **No automatic unit conversions.** The ESMF framework does not currently contain tools for performing unit conversions, operations that are fairly standard within Coupler Components.

4. **No accumulator.** The ESMF does not have an accumulator tool, to perform time averaging of fields for coupling. This is likely to be developed in the near term.

[2] It is not necessary to create a Coupler Component for each individual data *transfer.*

18.4 Class API

19 SciComp Class

19.1 Description

In Earth system modeling, a particular piece of code representing a physical domain, such as an atmospheric model or an ocean model, is typically implemented as an ESMF Gridded Component, or `ESMC_GridComp`. However, there are times when physical domains, or realms, need to be represented, but aren't actual pieces of code, or software. These domains can be implemented as ESMF Science Components, or `ESMC_SciComp`.

Unlike Gridded and Coupler Components, Science Components are not associated with software; they don't include execution routines such as initialize, run and finalize. The main purpose of a Science Component is to provide a container for Attributes within a Component hierarchy.

19.2 Use and Examples

A Science Component is a container object intended to represent scientific domains, or realms, in an Earth Science Model. It's primary purpose is to provide a means for representing Component metadata within a hierarchy of Components, and it does this by being a container for Attributes as well as other Components.

19.3 Restrictions and Future Work

1. **None.**

19.4 Class API

20 Fault-tolerant Component Tunnel

20.1 Description

For ensemble runs with many ensemble members, fault-tolerance becomes an issue of very critical practical impact. The meaning of *fault-tolerance* in this context refers to the ability of an ensemble application to continue with normal execution after one or more ensemble members have experienced catastrophic conditions, from which they cannot recover. ESMF implements this type of fault-tolerance on the Component level via a **timeout** paradigm: A timeout parameter is specified for all interactions that need to be fault-tolerant. When a connection to a component times out, maybe because it has become inaccessible due to some catastrophic condition, the driver application can react to this condition, for example by not further interacting with the component during the otherwise normal continuation of the model execution.

The fault-tolerant connection between a driver application and a Component is established through a **Component Tunnel**. There are two sides to a Component Tunnel: the "actual" side is where the component is actually executing, and the "dual" side is the portal through which the Component becomes accessible on the driver side. Both the actual and the dual side of a Component Tunnel are implemented in form of a regular ESMF Gridded or Coupler Component.

Component Tunnels between Components can be based on a number of low level implementations. The only implementation that currently provides fault-tolerance is *socket* based. In this case an actual Component typically runs as

a separate executable, listening to a specific port for connections from the driver application. The dual Component is created on the driver side. It connects to the actual Component during the SetServices() call.

20.2 Use and Examples

A Component Tunnel connects a *dual* Component to an *actual* Component. This connection can be based on a number of different low level implementations, e.g. VM-based or socket-based. VM-based Component Tunnels require that both dual and actual Components run within the same application (i.e. execute under the same MPI_COMM_WORLD). Fault-tolerant Component Tunnels require that dual and actual Components run in separate applications, under different MPI_COMM_WORLD communicators. This mode is implemented in the socket-based Component Tunnels.

20.3 Restrictions and Future Work

1. **No data flow through States.** The current implementation does not support data flow (Fields, FieldBundles, etc.) between actual and dual Components. The current work-around is to employ user controlled, file based transfer methods. The next implementation phase will offer transparent data flow through the Component Tunnel, where the user code interacts with the States on the actual and dual side in the same way as if they were the same Component.

21 State Class

21.1 Description

A State contains the data and metadata to be transferred between ESMF Components. It is an important class, because it defines a standard for how data is represented in data transfers between Earth science components. The State construct is a rational compromise between a fully prescribed interface - one that would dictate what specific fields should be transferred between components - and an interface in which data structures are completely ad hoc.

There are two types of States, import and export. An import State contains data that is necessary for a Gridded Component or Coupler Component to execute, and an export State contains the data that a Gridded Component or Coupler Component can make available.

States can contain Arrays, ArrayBundles, Fields, FieldBundles, and other States. They cannot directly contain native language arrays (i.e. Fortran or C style arrays). Objects in a State must span the VM on which they are running. For sequentially executing components which run on the same set of PETs this happens by calling the object create methods on each PET, creating the object in unison. For concurrently executing components which are running on subsets of PETs, an additional method, called `ESMF_StateReconcile()`, is provided by ESMF to broadcast information about objects which were created in sub-components.

State methods include creation and deletion, adding and retrieving data items, adding and retrieving attributes, and performing queries.

21.2 Constants

21.2.1 ESMF_STATEINTENT

DESCRIPTION:
Specifies whether a ESMF_State contains data to be imported into a component or exported from a component.

The type of this flag is:

type(ESMF_StateIntent_Flag)

The valid values are:

ESMF_STATEINTENT_IMPORT Contains data to be imported into a component.

ESMF_STATEINTENT_EXPORT Contains data to be exported out of a component.

ESMF_STATEINTENT_INTERNAL Contains data that is not exposed outside of a component.

ESMF_STATEINTENT_UNSPECIFIED The intent has not been specified.

21.2.2 ESMF_STATEITEM

DESCRIPTION:
Specifies the type of object being added to or retrieved from an ESMF_State.

The type of this flag is:

type(ESMF_StateItem_Flag)

The valid values are:

ESMF_STATEITEM_ARRAY Refers to an ESMF_Array within an ESMF_State.

ESMF_STATEITEM_ARRAYBUNDLE Refers to an ESMF_Array within an ESMF_State.

ESMF_STATEITEM_FIELD Refers to a ESMF_Field within an ESMF_State.

ESMF_STATEITEM_FIELDBUNDLE Refers to a ESMF_FieldBundle within an ESMF_State.

ESMF_STATEITEM_ROUTEHANDLE Refers to a ESMF_RouteHandle within an ESMF_State.

ESMF_STATEITEM_STATE Refers to a ESMF_State within an ESMF_State.

21.3 Use and Examples

A Gridded Component generally has one associated import State and one export State. Generally the States associated with a Gridded Component will be created by the Gridded Component's parent component. In many cases, the States will be created containing no data. Both the empty States and the newly created Gridded Component are passed by the parent component into the Gridded Component's initialize method. This is where the States get prepared for use and the import State is first filled with data.

States can be filled with data items that do not yet have data allocated. Fields, FieldBundles, Arrays, and ArrayBundles each have methods that support their creation without actual data allocation - the Grid and Attributes are set up but

no Fortran array of data values is allocated. In this approach, when a State is passed into its associated Gridded Component's initialize method, the incomplete Arrays, Fields, FieldBundles, and ArrayBundles within the State can allocate or reference data inside the initialize method.

States are passed through the interfaces of the Gridded and Coupler Components' run methods in order to carry data between the components. While we expect a Gridded Component's import State to be filled with data during initialization, its export State will typically be filled over the course of its run method. At the end of a Gridded Component's run method, the filled export State is passed out through the argument list into a Coupler Component's run method. We recommend the convention that it enters the Coupler Component as the Coupler Component's import State. Here the data is transformed into a form that another Gridded Component requires, and passed out of the Coupler Component as its export State. It can then be passed into the run method of a recipient Gridded Component as that component's import State.

While the above sounds complicated, the rule is simple: a State going into a component is an import State, and a State leaving a component is an export State.

Objects inside States are normally created in *unison* where each PET executing a component makes the same object create call. If the object contains data, like a Field, each PET may have a different local chunk of the entire dataset but each Field has the same name and is logically one part of a single distributed object. As States are passed between components, if any object in a State was not created in unison on all the current PETs then some PETs have no object to pass into a communication method (e.g. regrid or data redistribution). The `ESMF_StateReconcile()` method must be called to broadcast information about these objects to all PETs in a component; after which all PETs have a single uniform view of all objects and metadata.

If components are running in sequential mode on all available PETs and States are being passed between them there is no need to call `ESMF_StateReconcile` since all PETs have a uniform view of the objects. However, if components are running on a subset of the PETs, as is usually the case when running in concurrent mode, then when States are passed into components which contain a superset of those PETs, for example, a Coupler Component, all PETs must call `ESMF_StateReconcile` on the States before using them in any ESMF communication methods. The reconciliation process broadcasts information about objects which exist only on a subset of the PETs. On PETs missing those objects it creates a *proxy* object which contains any qualities of the original object plus enough information for it to be a data source or destination for a regrid or data redistribution operation.

21.4 Restrictions and Future Work

1. **No synchronization of object IDs at object create time - Unison Rule:** Object IDs are used during the reconcile process to identify objects which are unknown to some subset of the PETs in the currently running VM. Object IDs are assigned in sequential order at object create time across the context of the current VM without communication. This design was requested by the user community during ESMF object design to reduce communication and synchronization overhead when creating distributed ESMF objects. As a consequence it is required to create distributed ESMF objects in **unison** across all PETs of the current VM in order to keep the ESMF object identification in sync.

   Violation of the unison rule will lead to undefined behavior when reconciling a State that contains objects with inconsistent object IDs.

2. **Info keys on top level State not reconciled without actual objects present from the relevant sub-context.** One of the actions of the `ESMF_StateReconcile()` method is to reconcile the Info keys of the State object itself. The endresult is that the reconciled State has the same Info *keys* on all of the PETs of the VM across which it was reconciled – albeit with potentially different values across PETs (see the `ESMF_StateReconcile()` API doc for more details). An edge case for which `ESMF_StateReconcile()` does **not** provide Info key reconcilation is when keys were added under a component executing on a subset of PETs (compared to the

reconciling VM), but no actual object (Field, FieldBundle, Array, ArrayBundle, or nested State) was added under the VM of that sub-context.

The situation of unreconciled Info keys across PETs for an ESMF State is not an error condition per-se, however, it can lead to unexpected behavior in downstream code. Specifically if such code expects to find consistent Info keys across all PETs. If this is the case, care should be taken to ensure actual objects are added to the top level State on the sub-context PETs where new Info keys are added.

21.5 Design and Implementation Notes

1. States contain the name of the associated Component, a flag for Import or Export, and a list of data objects, which can be a combination of FieldBundles, Fields, and/or Arrays. The objects must be named and have the proper attributes so they can be identified by the receiver of the data. For example, units and other detailed information may need to be associated with the data as an Attribute.

2. Data contained in States must be created in unison on each PET of the current VM. This allows the creation process to avoid doing communications since each PET can compute any information it needs to know about any remote PET (for example, the grid distribute method can compute the decomposition of the grid on not only the local PET but also the remote PETs since it knows each PET is making the identical call). For all PETs to have a consistent view of the data this means objects must be given unique names when created, or all objects must be created in the same order on all PETs so ESMF can generate consistent default names for the objects.

   When running components on subsets of the original VM all the PETs can create consistent objects but then when they are put into a State and passed to a component with a different VM and a different set of PETs, a communication call (reconcile) must be made to communicate the missing information to the PETs which were not involved in the original object creation. The reconcile call broadcasts object lists; those PETs which are missing any objects in the total list can receive enough information to reconstruct a proxy object which contains all necessary information about that object, with no local data, on that PET. These proxy objects can be queried by ESMF routines to determine the amount of data and what PETs contain data which is destined to be moved to the local PET (for receiving data) and conversely, can determine which other PETs are going to receive data and how much (for sending data).

   For example, the FieldExcl system test creates 2 Gridded Components on separate subsets of PETs. They use the option of mapping particular, non-monotonic PETs to DEs. The following figures illustrate how the DEs are mapped in each of the Gridded Components in that test:

   In the coupler code, all PETs must make the reconcile call before accessing data in the State. On PETs which already contain data, the objects are unchanged. On PETs which were not involved during the creation of the FieldBundles or Fields, the reconcile call adds an object to the State which contains all the same metadata associated with the object, but creates a slightly different Grid object, called a Proxy Grid. These PETs contain no local data, so the Array object is empty, and the DELayout for the Grid is like this:

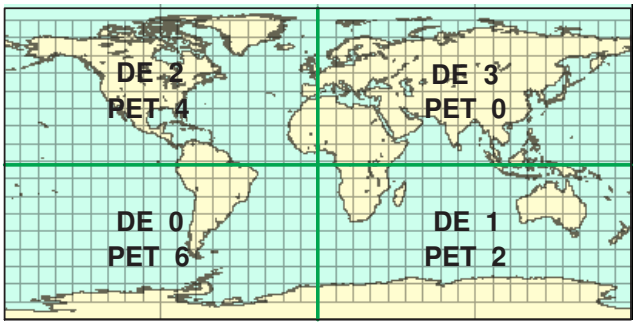


Figure 7: The mapping of PETs (processors) to DEs (data) in the source grid created by `user_model1.F90` in the FieldExcl system test.

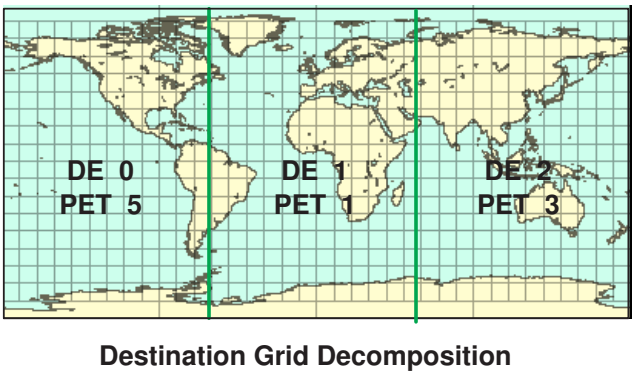
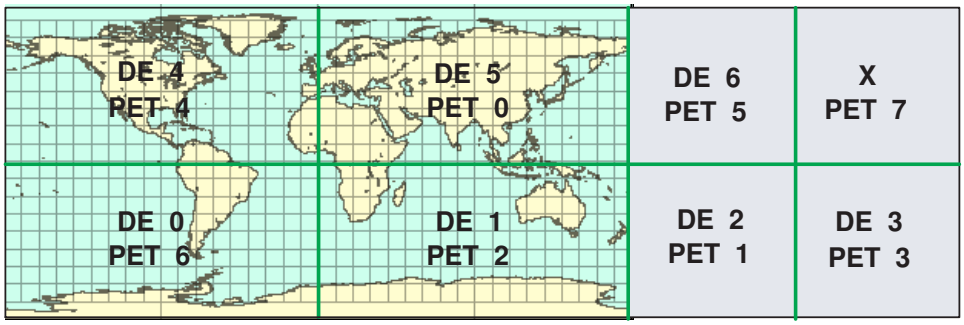
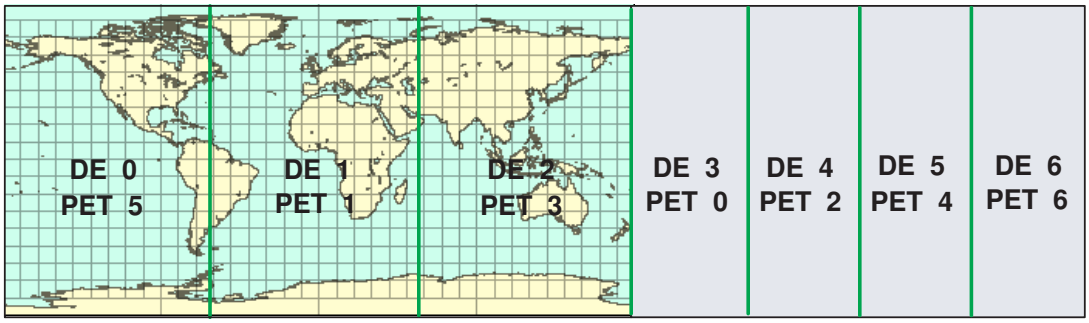


Figure 8: The mapping of PETs (processors) to DEs (data) in the destination grid created by `user_model2.F90` in the FieldExcl system test.



Proxy DELayout created by Framework for Source Grid Decomposition in Coupler

Figure 9: The mapping of PETs (processors) to DEs (data) in the source grid after the reconcile call in `user_coupler.F90` in the FieldExcl system test.

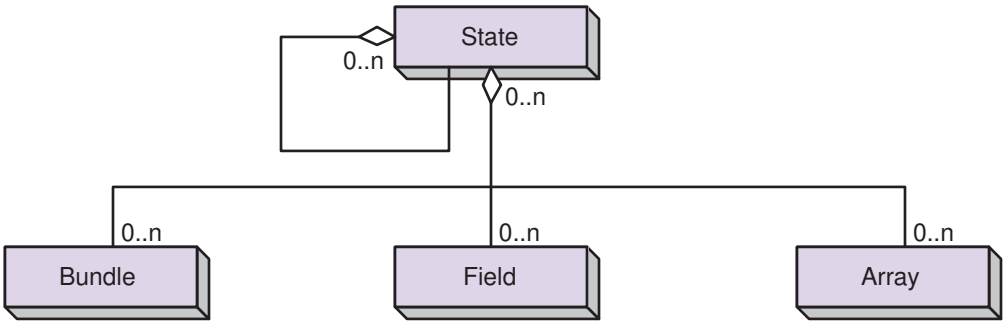


Proxy DELayout created by Framework for Destination Grid Decomposition in Coupler

Figure 10: The mapping of PETs (processors) to DEs (data) in the destination grid after the reconcile call in `user_coupler.F90` in the FieldExcl system test.

21.6 Object Model

The following is a simplified UML diagram showing the structure of the State class. States can contain FieldBundles, Fields, Arrays, or nested States. See Appendix A, *A Brief Introduction to UML*, for a translation table that lists the symbols in the diagram and their meaning.



21.7 Class API

22 Attachable Methods

22.1 Description

ESMF allows user methods to be attached to Components and States. Providing this capability supports a more object oriented way of model design.

Attachable methods on Components can be used to implement the concept of generic Components where the specialization requires attaching methods with well defined names. This methods are then called by the generic Component code.

Attaching methods to States can be used to supply data operations along with the data objects inside of a State object. This can be useful where a producer Component not only supplies a data set, but also the associated processing functionality. This can be more efficient than providing all of the possible sets of derived data.

22.2 Use and Examples

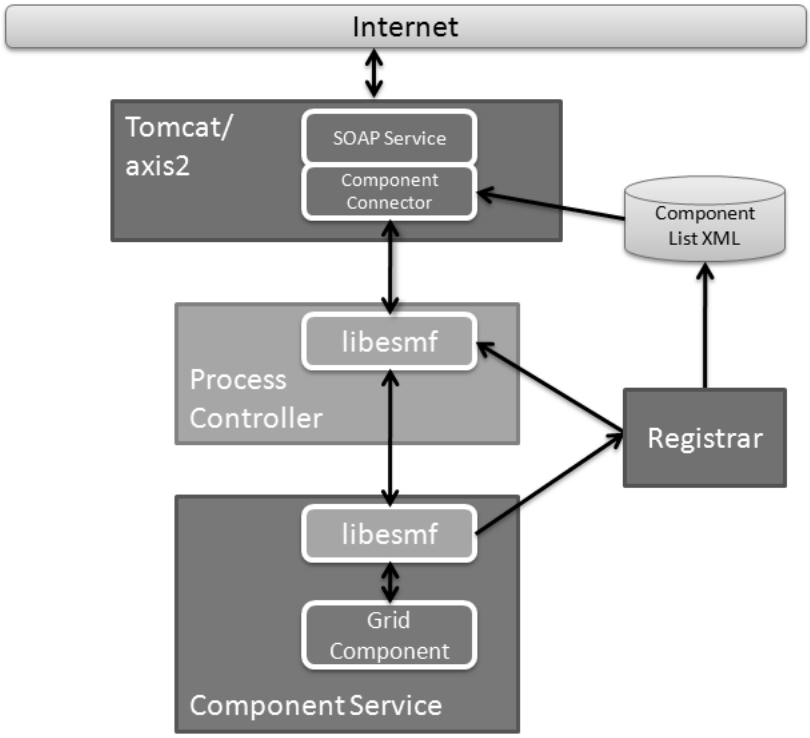
The following examples demonstrate how a producer Component attaches a user defined method to a State, and how it implements the method. The attached method is then executed by the consumer Component.

22.3 Restrictions and Future Work

1. **Not reconciled.** Attachable Methods are PET-local settings on an object. Currently Attachable Methods cannot be reconciled (i.e. ignored during `ESMF_StateReconcile()`).

2. **No copy nor move.** Currently Attachable Methods cannot be copied or moved between objects.

Figure 11: The diagram describes the ESMF Web Services software architecture. The architecture defines a multi-tiered set of applications that provide a flexible approach for accessing model components.



22.4 Class API

23 Web Services

23.1 Description

The goal of the ESMF Web Services is to provide the tools to allow ESMF Users to make their Components available via a web service. The first step is to make the Component a service, and then make it accessible via the Web.

At the heart of this architecture is the Component Service; this is the application that does the model work. The ESMF Web Services part provides a way to make the model accessible via a network API (Application Programming Interface). ESMF provides the tools to turn a model component into a service as well as the tools to access the service from the network.

The Process Controller is a stand-alone application that provides a control mechanism between the end user and the Component Service. The Process Controller is responsible for managing client information as well as restricting client access to a Component Service. (The role of the Process Controller is expected to expand in the future.)

The tomcat/axis2 application provides the access via the Web using standard SOAP protocols. Part of this application includes the SOAP interface definition (using a WSDL file) as well as some java code that provides the access to the Process Controller application.

Finally, the Registrar maintains a list of Component Services that are currently available; Component Services register themselves with the Registrar when they startup, and unregister themselves when they shutdown. The list of available services is maintained in an XML file and is accessible from the Registrar using its network API.

23.1.1 Creating a Service around a Component

23.1.2 Code Modifications

One of the goals in providing the tools to make Components into services was to make the process as simple and easy as possible. Any model component that has been implemented using the ESMF Component Framework can easily be turned into a Component Services with just a minor change to the Application driver code. (For details on the ESMF Framework, see the ESMF Developers Documentation.)

The primary function in ESMF Web Services is the ESMF_WebServicesLoop routine. This function registers the Component Service with the Registrar and then sets up a network socket service that listens for requests from a client. It starts a loop that waits for incoming requests and manages the routing of these requests to all PETs. It is also responsible for making sure the appropriate ESMF routine (ESMF_Initialize, ESMF_Run or ESMF_Finalize) is called based on the incoming request. When the client has completed its interaction with the Component Service, the loop will be terminated and it will unregister the Component Service from the Registrar.

To make all of this happen, the Application Driver just needs to replace its calls to ESMF_Initialize, ESMF_Run, and ESMF_Finalize with a single call to ESMF_WebServicesLoop.

```
use ESMF_WebServMod
....

call ESMF_WebServicesLoop(gridComponent, portNumber, returnCode)
```

That's all there is to turning an ESMF Component into a network-accessible ESMF Component Service. For a detailed example of an ESMF Component turned into an ESMF Component Service, see the Examples in the Web Services section of the Developer' Guide.

23.1.3 Accessing the Service

Now that the Component is available as a service, it can be accessed remotely by any client that can communicate via TCP sockets. The ESMF library, in addition to providing the service tools, also provides the classes to create C++ clients to access the Component Service via the socket interface.

However, the goal of ESMF Web Services is to make an ESMF Component accessible through a standard web service, which is accomplished through the Process Controller and the Tomcat/Axis2 applications

23.1.4 Client Application via C++ API

Interfacing to a Component service is fairly simple using the ESMF library. The following code is a simple example of how to interface to a Component Service in C++ and request the initialize operation (the entire sample client can be found in the Web Services examples section of the ESMF Distribution):

```
#include "ESMCI_WebServCompSvrClient.h"

int  main(int  argc, char*  argv[])
{
       int    portNum = 27060;
          int    clientId = 101;
       int    rc = ESMF_SUCCESS;

       ESMCI::ESMCI_WebServCompSvrClient
                        client("localhost", portNum, clientId);

       rc = client.init();
       printf("Initialize return code: %d\n", rc);
}
```

To see a complete description of the NetEsmfClient class, refer to the netesmf library section of the Web Services Reference Manual.

23.1.5 Process Controller

The Process Controller is basically just a instance of a C++ client application. It manages client access to the Component Service (only 1 client can access the service at a time), and will eventually be responsible for starting up and shutting down instances of Component Services (planned for a future release). The Process Controller application is built with the ESMF library and is included in the apps section of the distribution.

23.1.6 Tomcat/Axis2

The Tomcat/Axis2 "application" is essentially the Apache Tomcat server using the Apache Axis2 servlet to implement web services using SOAP protocols. The web interface is defined by a WSDL file, and its implementation is handled by the Component Connector java code. Tomcat and Axis2 are both open source projects that should be downloaded from the Apache web site, but the WSDL file, the Component Connector java code, and all required software for supporting the interface can be found next to the ESMF distribution in the web_services_server directory. This code is not included with the ESMF distribution because they can be distributed and installed independent of each other.

23.2 Use and Examples

The following examples demonstrate how to use ESMF Web Services.

23.3 Restrictions and Future Work

1. **Manual Control of Process.** Currently, the Component Service must be manually started and stopped. Future plans include having the Process Controller be responsible for controlling the Component Service processes.

2. **Data Streaming.** While data can be streamed from the web server to the client, it is not yet getting the data directly from the Component Service. Instead, the Component Service exports the data to a file which the

Process Controller can read and return across the network interface. The data streaming capabilities will be a major component of future improvements to the Web Services architecture.

23.4 Class API

Part IV

Infrastructure: Fields and Grids

24 Overview of Data Classes

The ESMF infrastructure data classes are part of the framework's hierarchy of structures for handling Earth system model data and metadata on parallel platforms. The hierarchy is in complexity; the simplest data class in the infrastructure represents a distributed data array and the most complex data class represents a bundle of physical fields that are discretized on the same grid. Data class methods are called both from user-written code and from other classes internal to the framework.

Data classes are distributed over **DE**s, or **Decomposition Elements**. A DE represents a piece of a decomposition. A DELayout is a collection of DEs with some associated connectivity that describes a specific distribution. For example, the distribution of a grid divided into four segments in the x-dimension would be expressed in ESMF as a DELayout with four DEs lying along an x-axis. This abstract concept enables a data decomposition to be defined in terms of threads, MPI processes, virtual decomposition elements, or combinations of these without changes to user code. This is a primary strategy for ensuring optimal performance and portability for codes using ESMF for communications.

ESMF data classes provide a standard, convenient way for developers to collect together information related to model or observational data. The information assembled in a data class includes a data pointer, a set of attributes (e.g. units, although attributes can also be user-defined), and a description of an associated grid. The same set of information within an ESMF data object can be used by the framework to arrange intercomponent data transfers, to perform I/O, for communications such as gathers and scatters, for simplification of interfaces within user code, for debugging, and for other functions. This unifies and organizes codes overall so that the user need not define different representations of metadata for the same field for I/O and for component coupling.

Since it is critical that users be able to introduce ESMF into their codes easily and incrementally, ESMF data classes can be created based on native Fortran pointers. Likewise, there are methods for retrieving native Fortran pointers from within ESMF data objects. This allows the user to perform allocations using ESMF, and to retrieve Fortran arrays later for optimized model calculations. The ESMF data classes do not have associated differential operators or other mathematical methods.

For flexibility, it is not necessary to build an ESMF data object all at once. For example, it's possible to create a field but to defer allocation of the associated field data until a later time.

Key Features
Hierarchy of data structures designed specifically for the Earth system domain and high performance, parallel computing.
Multi-use ESMF structures simplify user code overall.
Data objects support incremental construction and deferred allocation.
Native Fortran arrays can be associated with or retrieved from ESMF data objects, for ease of adoption, convenience, and performance.
A variety of operations are provided for manipulating data in data objects such as regridding, redistribution, halo communication, and sparse matrix multiply.

The main classes that are used for model and observational data manipulation are as follows:

- **Array** An ESMF Array contains a data pointer, information about its associated datatype, precision, and dimension.

  Data elements in Arrays are partitioned into categories defined by the role the data element plays in distributed halo operations. Haloing - sometimes called ghosting - is the practice of copying portions of array data to multiple memory locations to ensure that data dependencies can be satisfied quickly when performing a calculation. ESMF Arrays contain an **exclusive** domain, which contains data elements updated exclusively and definitively by a given DE; a **computational** domain, which contains all data elements with values that are updated by the

DE in computations; and a **total** domain, which includes both the computational domain and data elements from other DEs which may be read but are not updated in computations.

- **ArrayBundle** ArrayBundles are collections of Arrays that are stored in a single object. Unlike FieldBundles, they don't need to be distributed the same way across PETs. The motivation for ArrayBundles is both convenience and performance.

- **Field** A Field holds model and/or observational data together with its underlying grid or set of spatial locations. It provides methods for configuration, initialization, setting and retrieving data values, data I/O, data regridding, and manipulation of attributes.

- **FieldBundle** Groups of Fields on the same underlying physical grid can be collected into a single object called a FieldBundle. A FieldBundle provides two major functions: it allows groups of Fields to be manipulated using a single identifier, for example during export or import of data between Components; and it allows data from multiple Fields to be packed together in memory for higher locality of reference and ease in subsetting operations. Packing a set of Fields into a single FieldBundle before performing a data communication allows the set to be transferred at once rather than as a Field at a time. This can improve performance on high-latency platforms.

  FieldBundle objects contain methods for setting and retrieving constituent fields, regridding, data I/O, and reordering of data in memory.

24.1 Bit-for-Bit Considerations

Bit-for-bit reproducibility is at the core of the regression testing schemes of many scientific model codes. The bit-for-bit requirement makes it easy to compare the numerical results of simulation runs using standard binary diff tools.

For the most part, ESMF methods do not modify user data numerically, and thus have no effect on the bit-for-bit characteristics of the model code. The exceptions are the regrid weight generation and the sparse matrix multiplication.

In the case of the regrid weight generation, user data is used to produce interpolation weights following specific numerical schemes. The bit-for-bit reproducibility of the generated weights depends on the implementation details. Section 24.2 provides more details about the bit-for-bit considerations with respect to the regrid weights generated by ESMF.

In the case of the sparse matrix multiplication, which is the typical method that is used to apply the regrid weights, user data is directly manipulated by ESMF. In order to help users with the implementation of their bit-for-bit requirements, while also considering the associated performance impact, the ESMF sparse matrix implementation provides three levels of bit-for-bit support. The strictest level ensures that the numerical results are bit-for-bit identical, even when executing across different numbers of PETs. In the relaxed level, bit-for-bit reproducibility is guaranteed when running across an unchanged number of PETs. The lowest level makes no guarantees about bit-for-bit reproducibility, however, it provides the greatest performance potential for those cases where numerical round-off differences are acceptable. An in-depth discussion of bit-for-bit reproducibility, and the performance aspects of route-based communication methods, such the sparse matrix multiplication, is given in section **??**.

24.2 Regrid

This section describes the regridding methods provided by ESMF. Regridding, also called remapping or interpolation, is the process of changing the grid that underlies data values while preserving qualities of the original data. Different kinds of transformations are appropriate for different problems. Regridding may be needed when communicating data between Earth system model components such as land and atmosphere, or between different data sets to support operations such as visualization.

Regridding can be broken into two stages. The first stage is generation of an interpolation weight matrix that describes how points in the source grid contribute to points in the destination grid. The second stage is the multiplication of values on the source grid by the interpolation weight matrix to produce values on the destination grid. This is implemented as a parallel sparse matrix multiplication.

There are two options for accessing ESMF regridding functionality: **offline** and **integrated**. Offline regridding is a process whereby interpolation weights are generated by a separate ESMF command line tool, not within the user code. The ESMF offline regridding tool also only generates the interpolation matrix, the user is responsible for reading in this matrix and doing the actual interpolation (multiplication by the sparse matrix) in their code. Please see Section 12 for a description of the offline regridding command line tool and the options it supports. For user convenience, there is also a method interface to the offline regrid tool functionality which is described in Section **??**. In contrast to offline regridding, integrated regridding is a process whereby interpolation weights are generated via subroutine calls during the execution of the user's code. In addition to generating the weights, integrated regridding can also produce a **RouteHandle** (described in Section 37.1) which allows the user to perform the parallel sparse matrix multiplication using ESMF methods. In other words, ESMF integrated regridding allows a user to perform the whole process of interpolation within their code.

To see what types of grids and other options are supported in the two types of regridding and their testing status, please see the ESMF Regridding Status webpage for this version of ESMF. Figure 24.2 shows a comparison of different regrid interfaces and where they can be found in the documentation.

The rest of this section further describes the various options available in ESMF regridding.

Name	Access via	Inputs	Outputs		Description
			Weights	RouteHandle	
ESMF_FieldRegridStore()	Subroutine call	Field object	yes	yes	Sec. **??**
ESMF_FieldBundleRegridStore()	Subroutine call	Fieldbundle obj.	no	yes	Sec. **??**
ESMF_RegridWeightGen()	Subroutine call	Grid files	yes	no	Sec. **??**
ESMF_RegridWeightGen	Command Line Tool	Grid files	yes	no	Sec. 12

Table 1: Regrid Interfaces

24.2.1 Interpolation methods: bilinear

Bilinear interpolation calculates the value for the destination point as a combination of multiple linear interpolations, one for each dimension of the Grid. Note that for ease of use, the term bilinear interpolation is used for 3D interpolation in ESMF as well, although it should more properly be referred to as trilinear interpolation.

In 2D, ESMF supports bilinear regridding between any combination of the following:

- Structured grids (`ESMF_Grid`) composed of any number of logically rectangular tiles
- Unstructured meshes (`ESMF_Mesh`) composed of polygons with any number of sides
- A set of disconnected points (`ESMF_LocStream`) may be the destination of the regridding
- An exchange grid (`ESMF_XGrid`)

In 3D, ESMF supports bilinear regridding between any combination of the following:

- Structured grids (`ESMF_Grid`) composed of a single logically rectangular tile

- Unstructured meshes (ESMF_Mesh) composed of hexahedrons
- A set of disconnected points (ESMF_LocStream) may be the destination of the regridding

Restrictions:

- Cells which contain enough identical corners to collapse to a line or point are currently ignored
- Self-intersecting cells (e.g. a cell twisted into a bow tie) are not supported
- On a spherical grid, cells which contain an edge which extends more than half way around the sphere are not supported
- Source Fields built on a Grid which contains a DE of width less than 2 elements are not supported

To use the bilinear method the user may create their Fields on any stagger location (e.g. ESMF_STAGGERLOC_CENTER) for a Grid, or any Mesh location (e.g. ESMF_MESHLOC_NODE) for a Mesh. For either a Grid or a Mesh, the location upon which the Field is built must contain coordinates. This method will also work with a destination Field built on a LocStream that contains coordinates, or with a source or destination Field built on an XGrid.

24.2.2 Interpolation methods: higher-order patch

Patch (or higher-order) interpolation is the ESMF version of a technique called "patch recovery" commonly used in finite element modeling [16] [14]. It typically results in better approximations to values and derivatives when compared to bilinear interpolation. Patch interpolation works by constructing multiple polynomial patches to represent the data in a source cell. For 2D grids, these polynomials are currently 2nd degree 2D polynomials. One patch is constructed for each corner of the source cell, and the patch is constructed by doing a least squares fit through the data in the cells surrounding the corner. The interpolated value at the destination point is then a weighted average of the values of the patches at that point.

The patch method has a larger stencil than the bilinear, for this reason the patch weight matrix can be correspondingly larger than the bilinear matrix (e.g. for a quadrilateral grid the patch matrix is around 4x the size of the bilinear matrix). This can be an issue when performing a regrid operation close to the memory limit on a machine.

The patch method does not guarantee that after regridding the range of values in the destination field is within the range of values in the source field. For example, if the mininum value in the source field is 0.0, then it's possible that after regridding with the patch method, the destination field will contain values less than 0.0.

In 2D, ESMF supports patch regridding between any combination of the following:

- Structured Grids (ESMF_Grid) composed of a single logically rectangular tile
- Unstructured meshes (ESMF_Mesh) composed of polygons with any number of sides
- A set of disconnected points (ESMF_LocStream) may be the destination of the regridding
- An exchange grid (ESMF_XGrid)

In 3D, ESMF supports patch regridding between any combination of the following:

- NONE

Restrictions:

- Cells which contain enough identical corners to collapse to a line or point are currently ignored
- Self-intersecting cells (e.g. a cell twisted into a bow tie) are not supported
- On a spherical grid, cells which contain an edge which extends more than half way around the sphere are not supported
- Source Fields built on a Grid which contains a DE of width less than 2 elements are not supported

To use the patch method the user may create their Fields on any stagger location (e.g. `ESMF_STAGGERLOC_CENTER`) for a Grid, or any Mesh location (e.g. `ESMF_MESHLOC_NODE`) for a Mesh. For either a Grid or a Mesh, the location upon which the Field is built must contain coordinates. This method will also work with a destination Field built on a LocStream that contains coordinates, or with a source or destination Field built on an XGrid.

24.2.3 Interpolation methods: nearest source to destination

In nearest source to destination interpolation (`ESMF_REGRIDMETHOD_NEAREST_STOD`) each destination point is mapped to the closest source point. A given source point may map to multiple destination points, but no destination point will receive input from more than one source point. If two points are equally close, then the point with the smallest sequence index is arbitrarily used (i.e. the point which would have the smallest index in the weight matrix).

In 2D, ESMF supports nearest source to destination regridding between any combination of the following:

- Structured Grids (`ESMF_Grid`) composed of any number of logically rectangular tiles
- Unstructured meshes (`ESMF_Mesh`) composed of polygons with any number of sides
- A set of disconnected points (`ESMF_LocStream`)
- An exchange grid (`ESMF_XGrid`)

In 3D, ESMF supports nearest source to destination regridding between any combination of the following:

- Structured Grids (`ESMF_Grid`) composed of any number of logically rectangular tiles
- Unstructured Meshes (`ESMF_Mesh`) composed of hexahedrons (e.g. cubes) and tetrahedrons
- A set of disconnected points (`ESMF_LocStream`)

Restrictions:
NONE

To use the nearest source to destination method the user may create their Fields on any stagger location (e.g. `ESMF_STAGGERLOC_CENTER`) for a Grid, or any Mesh location (e.g. `ESMF_MESHLOC_NODE`) for a Mesh. For either a Grid or a Mesh, the location upon which the Field is built must contain coordinates. This method will also work with a source or destination Field built on a LocStream that contains coordinates, or when the source or destination Field is built on an XGrid.

24.2.4 Interpolation methods: nearest destination to source

In nearest destination to source interpolation (ESMF_REGRIDMETHOD_NEAREST_DTOS) each source point is mapped to the closest destination point. A given destination point may receive input from multiple source points, but no source point will map to more than one destination point. If two points are equally close, then the point with the smallest sequence index is arbitrarily used (i.e. the point which would have the smallest index in the weight matrix). Note that with this method the unmapped destination point detection currently doesn't work, so no error will be returned even if there are destination points that don't map to any source point.

In 2D, ESMF supports nearest destination to source regridding between any combination of the following:

- Structured Grids (ESMF_Grid) composed of any number of logically rectangular tiles
- Unstructured meshes (ESMF_Mesh) composed of polygons with any number of sides
- A set of disconnected points (ESMF_LocStream)
- An exchange grid (ESMF_XGrid)

In 3D, ESMF supports nearest destination to source regridding between any combination of the following:

- Structured Grids (ESMF_Grid) composed of any number of logically rectangular tiles
- Unstructured Meshes (ESMF_Mesh) composed of hexahedrons (e.g. cubes) and tetrahedrons
- A set of disconnected points (ESMF_LocStream)

Restrictions:

- The unmapped destination point detection doesn't currently work for this method. Even if there are unmapped points, no error will be returned.

To use the nearest destination to source method the user may create their Fields on any stagger location (e.g. ESMF_STAGGERLOC_CENTER) for a Grid, or any Mesh location (e.g. ESMF_MESHLOC_NODE) for a Mesh. For either a Grid or a Mesh, the location upon which the Field is built must contain coordinates. This method will also work with a source or destination Field built on a LocStream that contains coordinates, or when the source or destination Field is built on an XGrid.

24.2.5 Interpolation methods: first-order conservative

The goal of this method is to preserve the integral of the field across the interpolation from source to destination. (For a more in-depth description of what this preservation of the integral (i.e. conservation) means please see section 24.2.7.) In this method the value across each source cell is treated as a constant, so it will typically have a larger interpolation error than the bilinear or patch methods. The first-order method used here is similar to that described in the following paper [18].

In the first-order method, the values for a particular destination cell are a calculated as a combination of the values of the intersecting source cells. The weight of a given source cell's contribution to the total being the amount that that source cell overlaps with the destination cell. In particular, the weight is the ratio of the area of intersection of the source and destination cells to the area of the whole destination cell.

To see a description of how the different normalization options affect the values and integrals produced by the conservative methods see section 24.2.8. For Grids, Meshes, or XGrids on a sphere this method uses great circle cells, for a description of potential problems with these see 24.2.9.

In 2D, ESMF supports conservative regridding between any combination of the following:

- Structured Grids (`ESMF_Grid`) composed of any number of logically rectangular tiles
- Unstructured meshes (`ESMF_Mesh`) composed of polygons with any number of sides
- An exchange grid (`ESMF_XGrid`)

In 3D, ESMF supports conservative regridding between any combination of the following:

- Structured Grids (`ESMF_Grid`) composed of a single logically rectangular tile
- Unstructured Meshes (`ESMF_Mesh`) composed of hexahedrons (e.g. cubes) and tetrahedrons

Restrictions:

- Cells which contain enough identical corners to collapse to a line or point are optionally (via a flag) either ignored or return an error
- Self-intersecting cells (e.g. a cell twisted into a bow tie) are not supported
- On a spherical grid, cells which contain an edge which extends more than half way around the sphere are not supported
- Source or destination Fields built on a Grid which contains a DE of width less than 2 elements are not supported

To use the conservative method the user should create their Fields on the center stagger location (`ESMF_STAGGERLOC_CENTER` in 2D or `ESMF_STAGGERLOC_CENTER_VCENTER` in 3D) for Grids or the element location (`ESMF_MESHLOC_ELEMENT`) for Meshes. For Grids, the corner stagger location (`ESMF_STAGGERLOC_CORNER` in 2D or `ESMF_STAGGERLOC_CORNER_VFACE` in 3D) must contain coordinates describing the outer perimeter of the Grid cells. This method will also work when the source or destination Field is built on an XGrid.

24.2.6 Interpolation methods: second-order conservative

Like the first-order conservative method, this method's goal is to preserve the integral of the field across the interpolation from source to destination. (For a more in-depth description of what this preservation of the integral (i.e. conservation) means please see section 24.2.7.) The difference between the first and second-order conservative methods is that the second-order takes the source gradient into account, so it yields a smoother destination field that typically better matches the source field. This difference between the first and second-order methods is particularly apparent when going from a coarse source grid to a finer destination grid. Another difference is that the second-order method does not guarantee that after regridding the range of values in the destination field is within the range of values in the source field. For example, if the mininum value in the source field is 0.0, then it's possible that after regridding with the second-order method, the destination field will contain values less than 0.0. The implementation of this method is based on the one described in this paper [12].

Like the first-order method, the values for a particular destination cell with the second-order method are a combination of the values of the intersecting source cells with the weight of a given source cell's contribution to the total being

the amount that that source cell overlaps with the destination cell. However, with the second-order conservative interpolation there are additional terms that take into account the gradient of the field across the source cell. In particular, the value d for a given destination cell is calculated as:

$d = \sum_{i}^{intersecting-source-cells}(s_i + \nabla s_i \cdot (c_{si} - c_d))$

Where:

s_i is the intersecting source cell value.

∇s_i is the intersecting source cell gradient.

c_{si} is the intersecting source cell centroid.

c_d is the destination cell centroid.

To see a description of how the different normalization options affect the values and integrals produced by the conservative methods see section 24.2.8. For Grids, Meshes, or XGrids on a sphere this method uses great circle cells, for a description of potential problems with these see 24.2.9.

In 2D, ESMF supports second-order conservative regridding between any combination of the following:

- Structured Grids (`ESMF_Grid`) composed of any number of logically rectangular tiles
- Unstructured meshes (`ESMF_Mesh`) composed of polygons with any number of sides
- An exchange grid (`ESMF_XGrid`)

In 3D, ESMF supports second-order conservative regridding between any combination of the following:

- NONE

Restrictions:

- Cells which contain enough identical corners to collapse to a line or point are optionally (via a flag) either ignored or return an error
- Self-intersecting cells (e.g. a cell twisted into a bow tie) are not supported
- On a spherical grid, cells which contain an edge which extends more than half way around the sphere are not supported
- Source or destination Fields built on a Grid which contains a DE of width less than 2 elements are not supported

To use the second-order conservative method the user should create their Fields on the center stagger location (`ESMF_STAGGERLOC_CENTER` for Grids or the element location (`ESMF_MESHLOC_ELEMENT`) for Meshes. For Grids, the corner stagger location (`ESMF_STAGGERLOC_CORNER` in 2D must contain coordinates describing the outer perimeter of the Grid cells. This method will also work when the source or destination Field is built on an XGrid.

24.2.7 Conservation

Conservation means that the following equation will hold: $\sum^{all-source-cells}(V_{si} * A_{si}) = \sum^{all-destination-cells}(V_{dj} * A_{dj})$, where V is the variable being regridded and A is the area of a cell. The subscripts s and d refer to source and destination values, and the i and j are the source and destination grid cell indices (flattening the arrays to 1 dimension).

If the user doesn't specify a cell areas in the involved Grids or Meshes, then the areas (A) in the above equation are calculated by ESMF. For Cartesian grids, the area of a grid cell calculated by ESMF is the typical Cartesian area. For grids on a sphere, cell areas are calculated by connecting the corner coordinates of each grid cell with great circles. If the user does specify the areas in the Grid or Mesh, then the conservation will be adjusted to work for the areas provided by the user. This means that the above equation will hold, but with the areas (A) being the ones specified by the user.

The user should be aware that because of the conservation relationship between the source and destination fields, the more the total source area differs from the total destination area the more the values of the source field will differ from the corresponding values of the destination field, likely giving a higher interpolation error. It is best to have the total source and destination areas the same (this will automatically be true if no user areas are specified). For source and destination grids that only partially overlap, the overlapping regions of the source and destination should be the same.

24.2.8 The effect of normalization options on integrals and values produced by conservative methods

It is important to note that by default (i.e. using destination area normalization) conservative regridding doesn't normalize the interpolation weights by the destination fraction. This means that for a destination grid which only partially overlaps the source grid the destination field that is output from the regrid operation should be divided by the corresponding destination fraction to yield the true interpolated values for cells which are only partially covered by the source grid. The fraction also needs to be included when computing the total source and destination integrals. (To include the fraction in the conservative weights, the user can specify the fraction area normalization type. This can be done by specifying `normType=ESMF_NORMTYPE_FRACAREA` when invoking `ESMF_FieldRegridStore()`.)

For weights generated using destination area normalization (either by not specifying any normalization type or by specifying `normType=ESMF_NORMTYPE_DSTAREA`), if a destination field extends outside the unmasked source field, then the values of the cells which extend partway outside the unmasked source field are decreased by the fraction they extend outside. To correct these values, the destination field (`dst_field`) resulting from the `ESMF_FieldRegrid()` call can be divided by the destination fraction `dst_frac` from the `ESMF_FieldRegridStore()` call. The following pseudocode demonstrates how to do this:

```
for each destination element i
   if (dst_frac(i) not equal to 0.0) then
      dst_field(i)=dst_field(i)/dst_frac(i)
   end if
end for
```

For weights generated using destination area normalization (either by not specifying any normalization type or by specifying `normType=ESMF_NORMTYPE_DSTAREA`), the following pseudo-code shows how to compute the total destination integral (`dst_total`) given the destination field values (`dst_field`) resulting from the `ESMF_FieldRegrid()` call, the destination area (`dst_area`) from the `ESMF_FieldRegridGetArea()` call, and the destination fraction (`dst_frac`) from the `ESMF_FieldRegridStore()` call. As shown in the previous paragraph, it also shows how to adjust the destination field (`dst_field`) resulting from the `ESMF_FieldRegrid()` call by the fraction (`dst_frac`) from the `ESMF_FieldRegridStore()` call:

```
dst_total=0.0
for each destination element i
   if (dst_frac(i) not equal to 0.0) then
      dst_total=dst_total+dst_field(i)*dst_area(i)
      dst_field(i)=dst_field(i)/dst_frac(i)
      ! If mass computed here after dst_field adjust, would need to be:
      ! dst_total=dst_total+dst_field(i)*dst_area(i)*dst_frac(i)
   end if
end for
```

For weights generated using fraction area normalization (by specifying `normType=ESMF_NORMTYPE_FRACAREA`), no adjustment of the destination field is necessary. The following pseudo-code shows how to compute the total destination integral (`dst_total`) given the destination field values (`dst_field`) resulting from the `ESMF_FieldRegrid()` call, the destination area (`dst_area`) from the `ESMF_FieldRegridGetArea()` call, and the destination fraction (`dst_frac`) from the `ESMF_FieldRegridStore()` call:

```
dst_total=0.0
for each destination element i
     dst_total=dst_total+dst_field(i)*dst_area(i)*dst_frac(i)
end for
```

For both normalization types, the following pseudo-code shows how to compute the total source integral (`src_total`) given the source field values (`src_field`), the source area (`src_area`) from the `ESMF_FieldRegridGetArea()` call, and the source fraction (`src_frac`) from the `ESMF_FieldRegridStore()` call:

```
src_total=0.0
for each source element i
   src_total=src_total+src_field(i)*src_area(i)*src_frac(i)
end for
```

24.2.9 Great circle cells

For Grids, Meshes, or XGrids on a sphere some combinations of interpolation options (e.g. first and second-order conservative methods) use cells whose edges are great circles. This section describes some behavior that the user may not expect from these cells and some potential solutions.

A great circle edge isn't necessarily the same as a straight line in latitude longitude space. For small edges, this difference will be small, but for long edges it could be significant. This means if the user expects cell edges as straight lines in latitude longitude space, they should avoid using one large cell with long edges to compute an average over a region (e.g. over an ocean basin).

Also, the user should also avoid using cells that contain one edge that runs half way or more around the earth, because the regrid weight calculation assumes the edge follows the shorter great circle path. There isn't a unique great circle edge defined between points on the exact opposite side of the earth from one another (antipodal points). However, the user can work around both of these problem by breaking the long edge into two smaller edges by inserting an extra node, or by breaking the large target grid cells into two or more smaller grid cells. This allows the application to resolve the ambiguity in edge direction.

24.2.10 Masking

Masking is the process whereby parts of a Grid, Mesh, or LocStream can be marked to be ignored during an operation, such as when they are used in regridding. Masking can be used on a Field created from a regridding source to indicate that certain portions should not be used to generate regridded data. This is useful, for example, if a portion of the source contains unusable values. Masking can also be used on a Field created from a regridding destination to indicate that a certain portion should not receive regridded data. This is useful, for example, when part of the destination isn't being used (e.g. the land portion of an ocean grid).

The user may mask out points in the source Field or destination Field or both. To do masking the user sets mask information in the Grid (see **??**), Mesh (see **??**), or LocStream (see 32.2.2) upon which the Fields passed into the `ESMF_FieldRegridStore()` call are built. The `srcMaskValues` and `dstMaskValues` arguments to that call can then be used to specify which values in that mask information indicate that a location should be masked out. For example, if `dstMaskValues` is set to (/1,2/), then any location that has a value of 1 or 2 in the mask information of the Grid, Mesh or LocStream upon which the destination Field is built will be masked out.

Masking behavior differs slightly between regridding methods. For non-conservative regridding methods (e.g. bilinear or high-order patch), masking is done on points. For these methods, masking a destination point means that that point won't participate in regridding (e.g. won't be interpolated to). For these methods, masking a source point means that the entire source cell using that point is masked out. In other words, if any corner point making up a source cell is masked then the cell is masked. For conservative regridding methods (e.g. first-order conservative) masking is done on cells. Masking a destination cell means that the cell won't participate in regridding (e.g. won't be interpolated to). Similarly, masking a source cell means that the cell won't participate in regridding (e.g. won't be interpolated from). For any type of interpolation method (conservative or non-conservative) the masking is set on the location upon which the Fields passed into the regridding call are built. For example, if Fields built on `ESMF_STAGGERLOC_CENTER` are passed into the `ESMF_FieldRegridStore()` call then the masking should also be set on `ESMF_STAGGERLOC_CENTER`.

24.2.11 Extrapolation methods: overview

Extrapolation in the ESMF regridding system is a way to automatically fill some or all of the destination points left unmapped by a regridding method. Weights generated by the extrapolation method are merged into the regridding weights to yield one set of weights or routehandle. Currently extrapolation is not supported with conservative regridding methods, because doing so would result in non-conservative weights.

24.2.12 Extrapolation methods: nearest source to destination

In nearest source to destination extrapolation (`ESMF_EXTRAPMETHOD_NEAREST_STOD`) each unmapped destination point is mapped to the closest source point. A given source point may map to multiple destination points, but no destination point will receive input from more than one source point. If two points are equally close, then the point with the smallest sequence index is arbitrarily used (i.e. the point which would have the smallest index in the weight matrix).

If there is at least one unmasked source point, then this method is expected to fill all unmapped points.

24.2.13 Extrapolation methods: inverse distance weighted average

In inverse distance weighted average extrapolation (`ESMF_EXTRAPMETHOD_NEAREST_IDAVG`) each unmapped destination point is the weighted average of the closest N source points. The weight is the reciprocal of the distance

of the source point from the destination point raised to a power P. All the weights contributing to one destination point are normalized so that they sum to 1.0. The user can choose N and P when using this method, but defaults are also provided. For example, when calling `ESMF_FieldRegridStore()` N is specified via the argument `extrapNumSrcPnts` and P is specified via the argument `extrapDistExponent`.

If there is at least one unmasked source point, then this method is expected to fill all unmapped points.

24.2.14 Extrapolation methods: creep fill

In creep fill extrapolation (`ESMF_EXTRAPMETHOD_CREEP`) unmapped destination points are filled by repeatedly moving data from mapped locations to neighboring unmapped locations for a user specified number of levels. More precisely, for each creeped point, its value is the average of the values of the point's immediate neighbors in the previous level. For the first level, the values are the average of the point's immediate neighbors in the destination points mapped by the regridding method. The number of creep levels is specified by the user. For example, in ESMF_FieldRegridStore() this number of levels is specified via the `extrapNumLevels` argument.

Unlike some extrapolation methods, creep fill does not necessarily fill all unmapped destination points. Unfilled destination points are still unmapped with the usual consequences (e.g. they won't be in the resulting regridding matrix, and won't be set by the application of the regridding weights).

Because it depends on the connections in the destination grid, creep fill extrapolation is not supported when the destination Field is built on a Location Stream (ESMF_LocStream). Also, creep fill is currently only supported for 2D Grids, Meshes, or XGrids

24.2.15 Unmapped destination points

If a destination point can't be mapped to a location in the source grid by the combination of regrid method and optional follow on extrapolation method, then the user has two choices. The user may ignore those destination points that can't be mapped by setting the `unmappedaction` argument to `ESMF_UNMAPPEDACTION_IGNORE` (Ignored points won't be included in the sparse matrix or routeHandle). If the user needs the unmapped points, the `ESMF_FieldRegridStore()` method has the capability to return a list of them using the `unmappedDstList` argument. In addition to ignoring them, the user also has the option to return an error if unmapped destination points exist. This is the default behavior, so the user can either not set the `unmappedaction` argument or the user can set it to `ESMF_UNMAPPEDACTION_ERROR`. Currently, the unmapped destination error detection doesn't work with the nearest destination to source regrid method (`ESMF_REGRIDMETHOD_NEAREST_DTOS`), so with this method the regridding behaves as if `ESMF_UNMAPPEDACTION_IGNORE` is always on.

24.2.16 Spherical grids and poles

In the case that the Grid is on a sphere (`coordSys=ESMF_COORDSYS_SPH_DEG` or `ESMF_COORDSYS_SPH_RAD`) then the coordinates given in the Grid are interpreted as latitude and longitude values. The coordinates can either be in degrees or radians as indicated by the `coordSys` flag set during Grid creation. As is true with many global models, this application currently assumes the latitude and longitude refer to positions on a perfect sphere, as opposed to a more complex and accurate representation of the Earth's true shape such as would be used in a GIS system. (ESMF's current user base doesn't require this level of detail in representing the Earth's shape, but it could be added in the future if necessary.)

For Grids on a sphere, the regridding occurs in 3D Cartesian to avoid problems with periodicity and with the pole singularity. This library supports four options for handling the pole region (i.e. the empty area above the top row of

the source grid or below the bottom row of the source grid). Note that all of these pole options currently only work for the Fields build on the Grid class.

The first option is to leave the pole region empty (`polemethod=ESMF_POLEMETHOD_NONE`), in this case if a destination point lies above or below the top row of the source grid, it will fail to map, yielding an error (unless `unmappedaction=ESMF_UNMAPPEDACTION_IGNORE` is specified).

With the next two options (`ESMF_POLEMETHOD_ALLAVG` and `ESMF_POLEMETHOD_NPNTAVG`), the pole region is handled by constructing an artificial pole in the center of the top and bottom row of grid points and then filling in the region from this pole to the edges of the source grid with triangles. The pole is located at the average of the position of the points surrounding it, but moved outward to be at the same radius as the rest of the points in the grid. The difference between the two artificial pole options is what value is used at the pole. The option (`polemethod=ESMF_POLEMETHOD_ALLAVG`) sets the value at the pole to be the average of the values of all of the grid points surrounding the pole. The option (`polemethod=ESMF_POLEMETHOD_NPNTAVG`) allows the user to choose a number N from 1 to the number of source grid points around the pole. The value N is set via the argument `regridPoleNPnts`. For each destination point, the value at the pole is then the average of the N source points surrounding that destination point.

The last option (`polemethod=ESMF_POLEMETHOD_TEETH`) does not construct an artificial pole, instead the pole region is covered by connecting points across the top and bottom row of the source Grid into triangles. As this makes the top and bottom of the source sphere flat, for a big enough difference between the size of the source and destination pole regions, this can still result in unmapped destination points. Only pole option `ESMF_POLEMETHOD_NONE` is currently supported with the conservative interpolation methods (e.g. `regridmethod=ESMF_REGRIDMETHOD_CONSERVE`) and with the nearest neighbor interpolation options (e.g. `regridmethod=ESMF_REGRIDMETHOD_NEAREST_STOD`).

Regrid Method	Line Type	
	ESMF_LINETYPE_CART	ESMF_LINETYPE_GREAT_CIRCLE
ESMF_REGRIDMETHOD_BILINEAR	Y*	Y
ESMF_REGRIDMETHOD_PATCH	Y*	Y
ESMF_REGRIDMETHOD_NEAREST_STOD	Y*	N
ESMF_REGRIDMETHOD_NEAREST_DTOS	Y*	N
ESMF_REGRIDMETHOD_CONSERVE	N/A	Y*
ESMF_REGRIDMETHOD_CONSERVE_2ND	N/A	Y*

Table 2: Line Type Support by Regrid Method (* indicates the default)

Another variation in the regridding supported with spherical grids is **line type**. This is controlled in the `ESMF_FieldRegridStore()` method by the `lineType` argument. This argument allows the user to select the path of the line which connects two points on a sphere surface. This in turn controls the path along which distances are calculated and the shape of the edges that make up a cell. Both of these quantities can influence how interpolation weights are calculated, for example in bilinear interpolation the distances are used to calculate the weights and the cell edges are used to determine to which source cell a destination point should be mapped.

ESMF currently supports two line types: ESMF_LINETYPE_CART and ESMF_LINETYPE_GREAT_CIRCLE. The ESMF_LINETYPE_CART option specifies that the line between two points follows a straight path through the 3D Cartesian space in which the sphere is embedded. Distances are measured along this 3D Cartesian line. Under this option cells are approximated by planes in 3D space, and their boundaries are 3D Cartesian lines between their corner points. The ESMF_LINETYPE_GREAT_CIRCLE option specifies that the line between two points follows a great circle path along the sphere surface. (A great circle is the shortest path between two points on a sphere.) Distances are measured along the great circle path. Under this option cells are on the sphere surface, and their boundaries are great circle paths between their corner points.

Figure 24.2.16 shows which line types are supported for each regrid method as well as the defaults (indicated by *).

24.2.17 Vector regridding

ESMF's initial vector regridding capability is intended to give cleaner results for 2D spherical vectors expressed in terms of local directions (e.g. east and north) than regridding each vector component separately. To do this, it converts the vectors to 3D Cartesian space and then does the regridding there. This allows all the vectors participating in the regridding to have a consistent representation. After regridding, the resulting 3D vectors are then converted back to the local direction form. This entire process is expressed in the usual weight matrix and/or routeHandle form and so the typical `ESMF_FieldRegridStore()/ESMF_FieldRegrid()/ESMF_FieldRegridRelease()` regridding paradigm can be used. However, the weight matrix will be in the format that allows it to contain tensor dimension indices (i.e. the leading dimension of the `factorIndexList` will be of size 4).

In this initial version, the meaning of the different entries in the vector dimension are fixed. They will be interpreted as:

1st entry the east component of the vector

2nd entry the north component of the vector

Note that because the different components are mixed, using vector regridding with a conservative regrid method will not necessarily produce vectors whose components are conservative.

24.2.18 Troubleshooting guide

The below is a list of problems users commonly encounter with regridding and potential solutions. This is by no means an exhaustive list, so if none of these problems fit your case, or if the solutions don't fix your problem, please feel free to email esmf support (esmf_support@ucar.edu).

Problem: Regridding is too slow.

Possible Cause: The `ESMF_FieldRegridStore()` method is called more than is necessary.
The `ESMF_FieldRegridStore()` operation is a complex one and can be relatively slow for some cases (large Grids, 3D grids, etc.)

Solution: Reduce the number of `ESMF_FieldRegridStore()` calls to the minimum necessary. The routeHandle generated by the `ESMF_FieldRegridStore()` call depends on only four factors: the stagger locations that the input Fields are created on, the coordinates in the Grids the input Fields are built on at those stagger locations, the padding of the input Fields (specified by the `totalWidth` arguments in `FieldCreate`) and the size of the tensor dimensions in the input Fields (specified by the `ungridded` arguments in `FieldCreate`). For any pair of Fields which share these attributes with the Fields used in the `ESMF_FieldRegridStore` call the same routeHandle can be used. Note that the data in the Fields does NOT matter, the same routeHandle can be used no matter how the data in the Fields changes.

In particular:

- If Grid coordinates do not change during a run, then the `ESMF_FieldRegridStore()` call can be done once between a pair of Fields at the beginning and the resulting routeHandle used for each timestep during the run.

- If a pair of Fields was created with exactly the same arguments to `ESMF_FieldCreate()` as the pair of Fields used during an `ESMF_FieldRegridStore()` call, then the resulting routeHandle can also be used between that pair of Fields.

Problem: Distortions in destination Field at periodic boundary.

Possible Cause: The Grid overlaps itself. With a periodic Grid, the regrid system expects the first point to not be a repeat of the last point. In other words, regrid constructs its own connection and overlap between the first and last points of the periodic dimension and so the Grid doesn't need to contain these. If the Grid does, then this can cause problems.

Solution: Define the Grid so that it doesn't contain the overlap point. This typically means simply making the Grid one point smaller in the periodic dimension. If a Field constructed on the Grid needs to contain these overlap points then the user can use the `totalWidth` arguments to include this extra padding in the Field. Note, however, that the regrid won't update these extra points, so the user will have to do a copy to fill the points in the overlap region in the Field.

24.2.19 Restrictions and Future Work

This section contains restrictions that apply to the entire regridding system. For restrictions that apply to just one interpolation method, see the section corresponding to that method above.

- **Regridding doesn't work on a Field created on a Grid with an arbitrary distribution:** Using a Field built on a Grid with an arbitrary distribution will cause the regridding to stop with an error.

24.2.20 Design and implementation notes

The ESMF regrid weight calculation functionality has been designed to enable it to support a wide range of grid and interpolation types without needing to support each individual combination of source grid type, destination grid type, and interpolation method. To avoid the quadratic growth of the number of pairs of grid types, all grids are converted to a common internal format and the regrid weight calculation is performed on that format. This vastly reduces the variety of grids that need to be supported in the weight calculations for each interpolation method. It also has the added benefit of making it straightforward to add new grid types and to allow them to work with all the existing grid types. To hook into the existing weight calculation code, the new type just needs to be converted to the internal format.

The internal grid format used by the ESMF regrid weight calculation is a finite element unstructured mesh. This was chosen because it was the most general format and all the others could be converted to it. The ESMF finite element unstructured mesh (ESMF FEM) is similar in some respects to the SIERRA [13] package developed at Sandia National Laboratory. The ESMF code relies on some of the same underlying toolkits (e.g. Zoltan [11] library for calculating mesh partitions) and adds a layer on top that allows the calculation of regrid weights and some mesh operations (e.g. mesh redistribution) that ESMF needs. The ESMF FEM has similar notions to SIERRA about the basic structure of the mesh entities, fields, iteration and a similar notion of parallel distribution.

Currently we use the ESMF FEM internal mesh to hold the structure of our Mesh class and in our regrid weight calculation. The parts of the internal FEM code that are used/tested by ESMF are the following:

- The creation of a mesh composed of triangles and quadrilaterals or hexahedrons and tetrahedrons.
- The object relations data base to store the connections between objects (e.g. which element contains which nodes).

- The fields to hold data (e.g. coordinates). We currently only build fields on nodes and elements (2D and 3D).
- Iteration to move through mesh entities.
- The parallel code to maintain information about the distribution of the mesh across processors and to communicate data between parts of the mesh on different processors (i.e. halos).

24.3 File-based Regrid API

24.4 Restrictions and Future Work

1. **32-bit index limitation:** Currently all index space dimensions in an ESMF object are represented by signed 32-bit integers. This limits the number of elements in one-dimensional objects to the 32-bit limit. This limit can be crossed by higher dimensional objects, where the product space is only limited by the 64-bit sequence index representation.

25 FieldBundle Class

25.1 Description

A FieldBundle functions mainly as a convenient container for storing similar Fields. It represents "bundles" of Fields that are discretized on the same Grid, Mesh, LocStream, or XGrid and distributed in the same manner. The FieldBundle is an important data structure because it can be added to a State, which is used for sending and receiving data between Components.

In the common case where FieldBundle is built on top of a Grid, Fields within a FieldBundle may be located at different locations relative to the vertices of their common Grid. The Fields in a FieldBundle may be of different dimensions, as long as the Grid dimensions that are distributed are the same. For example, a surface Field on a distributed lat/lon Grid and a 3D Field with an added vertical dimension on the same distributed lat/lon Grid can be included in the same FieldBundle.

FieldBundles can be created and destroyed, can have Attributes added or retrieved, and can have Fields added, removed, replaced, or retrieved. Methods include queries that return information about the FieldBundle itself and about the Fields that it contains. The Fortran data pointer of a Field within a FieldBundle can be obtained by first retrieving the Field with a call to `ESMF_FieldBundleGet()`, and then using `ESMF_FieldGet()` to get the data.

In the future FieldBundles will serve as a mechanism for performance optimization. ESMF will take advantage of the similarities of the Fields within a FieldBundle to optimize collective communication, I/O, and regridding. See Section 25.3 for a description of features that are scheduled for future work.

25.2 Use and Examples

Examples of creating, accessing and destroying FieldBundles and their constituent Fields are provided in this section, along with some notes on FieldBundle methods.

25.3 Restrictions and Future Work

1. **No enforcement of the *same* Grid, Mesh, LocStream, or XGrid restriction.** While the documentation indicates in several places (including the Design and Implementation Notes) that a FieldBundle can only contain Fields that are built on the same Grid, Mesh, LocStream, or XGrid, and all Fields must have the same distribution, the actual FieldBundle implementation is more general and supports bundling of any Fields. The documentation, however, is in line with the long term plan of making the restrictive FieldBundle definition the default behavior. The more general bundling option would then be retained as a special case that requires explicit specification by the user. There is currently no functional difference in the FieldBundle implementation that profits from the documented restrictive approach. In addition, the general bundling option has been supported for a long time. Note however that the documented restrictive behavior is the anticipated long term default for FieldBundles.

2. **No mathematical operators.** The FieldBundle class does not support differential or other mathematical operators. We do not anticipate providing this functionality in the near future.

3. **Limited validation and print options.** We are planning to increase the number of validity checks available for FieldBundles as soon as possible. We also will be working on print options.

4. **Packed data has limited supported.** One of the options that we are currently working on for FieldBundles is packing. Packing means that the data from all the Fields that comprise the FieldBundle are manipulated collectively. This operation can be done without destroying the original Field data. Packing is being designed to

facilitate optimized regridding, data communication, and I/O operations. This will reduce the latency overhead of the communication.

CAUTION: For communication methods, the undistributed dimension representing the number of fields must have identical size between source and destination packed data. Communication methods do not permute the order of fields in the source and destination packed FieldBundle.

5. **Interleaving Fields within a FieldBundle.** Data locality is important for performance on some computing platforms. An interleave option will be added to allow the user to create a packed FieldBundle in which Fields are either concatenated in memory or in which Field elements are interleaved.

25.4 Design and Implementation Notes

1. **Fields in a FieldBundle reference the same Grid, Mesh, LocStream, or XGrid.** In order to reduce memory requirements and ensure consistency, the Fields within a FieldBundle all reference the same Grid, Mesh, LocStream, or XGrid object. This restriction may be relaxed in the future.

25.5 Class API: Basic FieldBundle Methods

26 Field Class

26.1 Description

An ESMF Field represents a physical field, such as temperature. The motivation for including Fields in ESMF is that bundles of Fields are the entities that are normally exchanged when coupling Components.

The ESMF Field class contains distributed and discretized field data, a reference to its associated grid, and metadata. The Field class stores the grid *staggering* for that physical field. This is the relationship of how the data array of a field maps onto a grid (e.g. one item per cell located at the cell center, one item per cell located at the NW corner, one item per cell vertex, etc.). This means that different Fields which are on the same underlying ESMF Grid but have different staggerings can share the same Grid object without needing to replicate it multiple times.

Fields can be added to States for use in inter-Component data communications. Fields can also be added to FieldBundles, which are groups of Fields on the same underlying Grid. One motivation for packing Fields into FieldBundles is convenience; another is the ability to perform optimized collective data transfers.

Field communication capabilities include: data redistribution, regridding, scatter, gather, sparse-matrix multiplication, and halo update. These are discussed in more detail in the documentation for the specific method calls. ESMF does not currently support vector fields, so the components of a vector field must be stored as separate Field objects.

26.1.1 Operations

The Field class allows the user to easily perform a number of operations on the data stored in a Field. This section gives a brief summary of the different types of operations and the range of their capabilities. The operations covered here are: redistribution (`ESMF_FieldRedistStore()`), sparse matrix multiply (`ESMF_FieldSMMStore()`), and regridding (`ESMF_FieldRegridStore()`).

The redistribution operation (`ESMF_FieldRedistStore()`) allows the user to move data between two Fields with the same size, but different distribution. This operation is useful, for example, to move data between two components with different distributions. Please see Section **??** for an example of the redistribution capability.

The sparse matrix multiplication operation (ESMF_FieldSMMStore()) allows the user to multiply the data in a Field by a sparse matrix. This operation is useful, for example, if the user has an interpolation matrix and wants to apply it to the data in a Field. Please see Section **??** for an example of the sparse matrix multiply capability.

The regridding operation (ESMF_FieldRegridStore()) allows the user to move data from one grid to another while maintaining certain properties of the data. Regridding is also called interpolation or remapping. In the Field regridding operation the grids the data is being moved between are the grids associated with the Fields storing the data. The regridding operation works on Fields built on Meshes, Grids, or Location Streams. There are six regridding methods available: bilinear, higher-order patch, two types of nearest neighbor, first-order conservative, and second-order conservative. Please see section 24.2 for a more indepth description of regridding including in which situations each method is supported. Please see section **??** for a description of the regridding capability as it applies to Fields. Several sections following section **??** contain examples of using regridding.

26.2 Constants

26.2.1 ESMF_FIELDSTATUS

DESCRIPTION:

An ESMF_Field can be in different status after initialization. Field status can be queried using ESMF_FieldGet() method.

The type of this flag is:

type(ESMF_FieldStatus_Flag)

The valid values are:

ESMF_FIELDSTATUS_EMPTY Field is empty without geombase or data storage. Such a Field can be added to a ESMF_State and participate ESMF_StateReconcile().

ESMF_FIELDSTATUS_GRIDSET Field is partially created. It has a geombase object internally created and the geombase object associates with either a ESMF_Grid, or a ESMF_Mesh, or an ESMF_XGrid, or a ESMF_LocStream. It's an error to set another geombase object in such a Field. It can also be added to a ESMF_State and participate ESMF_StateReconcile().

ESMF_FIELDSTATUS_COMPLETE Field is completely created with geombase and data storage internally allocated.

26.3 Use and Examples

A Field serves as an annotator of data, since it carries a description of the grid it is associated with and metadata such as name and units. Fields can be used in this capacity alone, as convenient, descriptive containers into which arrays can be placed and retrieved. However, for most codes the primary use of Fields is in the context of import and export States, which are the objects that carry coupling information between Components. Fields enable data to be self-describing, and a State holding ESMF Fields contains data in a standard format that can be queried and manipulated.

The sections below go into more detail about Field usage.

26.3.1 Field create and destroy

Fields can be created and destroyed at any time during application execution. However, these Field methods require some time to complete. We do not recommend that the user create or destroy Fields inside performance-critical computational loops.

All versions of the `ESMF_FieldCreate()` routines require a Grid object as input, or require a Grid be added before most operations involving Fields can be performed. The Grid contains the information needed to know which Decomposition Elements (DEs) are participating in the processing of this Field, and which subsets of the data are local to a particular DE.

The details of how the create process happens depend on which of the variants of the `ESMF_FieldCreate()` call is used. Some of the variants are discussed below.

There are versions of the `ESMF_FieldCreate()` interface which create the Field based on the input Grid. The ESMF can allocate the proper amount of space but not assign initial values. The user code can then get the pointer to the uninitialized buffer and set the initial data values.

Other versions of the `ESMF_FieldCreate()` interface allow user code to attach arrays that have already been allocated by the user. Empty Fields can also be created in which case the data can be added at some later time.

For versions of Create which do not specify data values, user code can create an ArraySpec object, which contains information about the typekind and rank of the data values in the array. Then at Field create time, the appropriate amount of memory is allocated to contain the data which is local to each DE.

When finished with a `ESMF_Field`, the `ESMF_FieldDestroy` method removes it. However, the objects inside the `ESMF_Field` created externally should be destroyed separately, since objects can be added to more than one `ESMF_Field`. For example, the same `ESMF_Grid` can be referenced by multiple `ESMF_Field`s. In this case the internal Grid is not deleted by the `ESMF_FieldDestroy` call.

26.4 Restrictions and Future Work

1. **CAUTION:** It depends on the specific entry point of `ESMF_FieldCreate()` used during Field creation, which Fortran operations are supported on the Fortran array pointer `farrayPtr`, returned by `ESMF_FieldGet()`. Only if the `ESMF_FieldCreate()` *from pointer* variant was used, will the returned `farrayPtr` variable contain the original bounds information, and be suitable for the Fortran `deallocate()` call. This limitation is a direct consequence of the Fortran 95 standard relating to the passing of array arguments.

2. **No mathematical operators.** The Fields class does not currently support advanced operations on fields, such as differential or other mathematical operators.

26.5 Design and Implementation Notes

1. Some methods which have a Field interface are actually implemented at the underlying Grid or Array level; they are inherited by the Field class. This allows the user API (Application Programming Interface) to present functions at the level which is most consistent to the application without restricting where inside the ESMF the actual implementation is done.

2. The Field class is implemented in Fortran, and as such is defined inside the framework by a Field derived type and a set of subprograms (functions and subroutines) which operate on that derived type. The Field class itself is very thin; it is a container class which groups a Grid and an Array object together.

3. Fields follow the framework-wide convention of the *unison* creation and operation rule: All PETs which are part of the currently executing VM must create the same Fields at the same point in their execution. Since an early user request was that global object creation not impose the overhead of a barrier or synchronization point, Field creation does no inter-PET communication. For this to work, each PET must query the total number of PETs in this VM, and which local PET number it is. It can then compute which DE(s) are part of the local decomposition, and any global information can be computed in unison by all PETs independently of the others. In this way the overhead of communication is avoided, at the cost of more difficulty in diagnosing program bugs which result from not all PETs executing the same create calls.

4. Related to the item above, the user request to not impose inter-PET communication at object creation time means that requirement FLD 1.5.1, that all Fields will have unique names, and if not specified, the framework will generate a unique name for it, is difficult or impossible to support. A part of this requirement has been implemented; a unique object counter is maintained in the Base object class, and if a name is not given at create time a name such as "Field003" is generated which is guaranteed to not be repeated by the framework. However, it is impossible to error check that the user has not replicated a name, and it is possible under certain conditions that if not all PETs have created the same number of objects, that the counters on different PETs may not stay synchronized. This remains an open issue.

26.6 Class API

26.7 Class API: Field Utilities

27 ArrayBundle Class

27.1 Description

The `ESMF_ArrayBundle` class allows a set of Arrays to be bundled into a single object. The Arrays in an ArrayBundle may be of different type, kind, rank and distribution. Besides ease of use resulting from bundling, the ArrayBundle class offers the opportunity for performance optimization when operating on a bundle of Arrays as a single entity. Communication methods are especially good candidates for performance optimization. Best optimization results are expected for ArrayBundles that contain Arrays that share a common distribution, i.e. DistGrid, and are of same type, kind and rank.

ArrayBundles are one of the data objects that can be added to States, which are used for providing to or receiving data from other Components.

27.2 Use and Examples

Examples of creating, destroying and accessing ArrayBundles and their constituent Arrays are provided in this section, along with some notes on ArrayBundle methods.

27.3 Restrictions and Future Work

- **Non-blocking** ArrayBundle communications option is not yet implemented. In the future this functionality will be provided via the `routesyncflag` option.

27.4 Design and Implementation Notes

The following is a list of implementation specific details about the current ESMF ArrayBundle.

- Implementation language is C++.
- All precomputed communication methods are based on sparse matrix multiplication.

27.5 Class API

28 Array Class

28.1 Description

The Array class is an alternative to the Field class for representing distributed, structured data. Unlike Fields, which are built to carry grid coordinate information, Arrays only carry information about the *indices* associated with grid cells. Since they do not have coordinate information, Arrays cannot be used to calculate interpolation weights. However, if the user supplies interpolation weights, the Array sparse matrix multiply (SMM) operation can be used to apply the weights and transfer data to the new grid. Arrays carry enough information to perform redistribution, scatter, and gather communication operations.

Like Fields, Arrays can be added to a State and used in inter-Component data communications. Arrays can also be grouped together into ArrayBundles, allowing operations to be performed collectively on the whole group. One motivation for this is convenience; another is the ability to schedule optimized, collective data transfers.

From a technical standpoint, the ESMF Array class is an index space based, distributed data storage class. Its purpose is to hold distributed user data. Each decompositon element (DE) is associated with its own memory allocation. The index space relationship between DEs is described by the ESMF DistGrid class. DEs, and their associated memory allocation, are pinned either to a specific perisistent execution thread (PET), virtual address space (VAS), or a single system image (SSI). This aspect is managed by the ESMF DELayout class. Pinning to PET is the most common mode and is the default.

The Array class offers common communication patterns within the index space formalism. All RouteHandle based communication methods of the Field, FieldBundle, and ArrayBundle layers are implemented via the Array SMM operation.

28.2 Use and Examples

An `ESMF_Array` is a distributed object that must exist on all PETs of the current context. Each PET-local instance of an Array object contains memory allocations for all PET-local DEs. There may be 0, 1, or more DEs per PET and the number of DEs per PET can differ between PETs for the same Array object. Memory allocations may be provided for each PET by the user during Array creation or can be allocated as part of the Array create call. Many of the concepts of the `ESMF_Array` class are illustrated by the following examples.

28.3 Restrictions and Future Work

- **CAUTION:** Depending on the specific `ESMF_ArrayCreate()` entry point used during Array creation, certain Fortran operations are not supported on the Fortran array pointer `farrayPtr`, returned by

ESMF_ArrayGet(). Only if the ESMF_ArrayCreate() *from pointer* variant was used, will the returned farrayPtr variable contain the original bounds information, and be suitable for the Fortran deallocate() call. This limitation is a direct consequence of the Fortran 95 standard relating to the passing of array arguments. Fortran array pointers returned from an Array that was created through the *assumed shape array* variant of ESMF_ArrayCreate() will have bounds that are consistent with the other arguments specified during Array creation. These pointers are not suitable for deallocation in accordance to the Fortran 95 standard.

- **1D limit:** ArrayHalo(), ArrayRedist() and ArraySMM() operations on Arrays created on DistGrids with arbitrary sequence indices are currently limited to 1D arbitrary DistGrids. There is no restriction on the number, size and mapping of undistributed Array dimensions in the presence of such a 1D arbitrary DistGrid.

28.4 Design and Implementation Notes

The Array class is part of the ESMF index space layer and is built on top of the DistGrid and DELayout classes. The DELayout class introduces the notion of *decomposition elements* (DEs) and their layout across the available PETs. The DistGrid describes how index space is decomposed by assigning *logically rectangular index space pieces* or *DE-local tiles* to the DEs. The Array finally associates a *local memory allocation* with each local DE.

The following is a list of implementation specific details about the current ESMF Array.

- Implementation language is C++.
- Local memory allocations are internally held in ESMF_LocalArray objects.
- All precomputed communication methods are based on sparse matrix multiplication.

28.5 Class API

28.6 Class API: DynamicMask Methods

29 LocalArray Class

29.1 Description

The ESMF_LocalArray class provides a language independent representation of data in array format. One of the major functions of the LocalArray class is to bridge the Fortran/C/C++ language difference that exists with respect to array representation. All ESMF Field and Array data is internally stored in ESMF LocalArray objects allowing transparent access from Fortran and C/C++.

In the ESMF Fortran API the LocalArray becomes visible in those cases where a local PET may be associated with multiple pieces of an Array, e.g. if there are multiple DEs associated with a single PET. The Fortran language standard does not provide an array of arrays construct, however arrays of derived types holding arrays are possible. ESMF calls use arguments that are of type `ESMF_LocalArray` with `dimension` attributes where necessary.

29.2 Restrictions and Future Work

- The TKR (type/kind/rank) overloaded LocalArray interfaces declare the dummy Fortran array arguments with the pointer attribute. The advantage of doing this is that it allows ESMF to inquire information about the provided Fortran array. The disadvantage of this choice is that actual Fortran arrays passed into these interfaces *must* also be defined with pointer attribute in the user code.

29.3 Class API

---

30 ArraySpec Class

30.1 Description

An ArraySpec is a very simple class that contains type, kind, and rank information about an Array. This information is stored in two parameters. **TypeKind** describes the data type of the elements in the Array and their precision. **Rank** is the number of dimensions in the Array.

The only methods that are associated with the ArraySpec class are those that allow you to set and retrieve this information.

30.2 Use and Examples

The ArraySpec is passed in as an argument at Field and FieldBundle creation in order to describe an Array that will be allocated or attached at a later time. There are any number of situations in which this approach is useful. One common example is a case in which the user wants to create a very flexible export State with many diagnostic variables predefined, but only a subset desired and consequently allocated for a particular run.

30.3 Restrictions and Future Work

1. **Limit on rank.** The values for type, kind and rank passed into the ArraySpec class are subject to the same limitations as Arrays. The maximum array rank is 7, which is the highest rank supported by Fortran.

30.4 Design and Implementation Notes

The information contained in an `ESMF_ArraySpec` is used to create `ESMF_Array` objects.

ESMF_ArraySpec is a shallow class, and only set and get methods are needed. They do not need to be created or destroyed.

30.5 Class API

31 Grid Class

31.1 Description

The ESMF Grid class is used to describe the geometry and discretization of logically rectangular physical grids. It also contains the description of the grid's underlying topology and the decomposition of the physical grid across the available computational resources. The most frequent use of the Grid class is to describe physical grids in user code so that sufficient information is available to perform ESMF methods such as regridding.

Key Features
Representation of grids formed by logically rectangular regions, including uniform and rectilinear grids (e.g. lat-lon grids), curvilinear grids (e.g. displaced pole grids), and grids formed by connected logically rectangular regions (e.g. cubed sphere grids).
Support for 1D, 2D, 3D, and higher dimension grids.
Distribution of grids across computational resources for parallel operations - users set which grid dimensions are distributed.
Grids can be created already distributed, so that no single resource needs global information during the creation process.
Options to define periodicity and other edge connectivities either explicitly or implicitly via shape shortcuts.
Options for users to define grid coordinates themselves or to call prefabricated coordinate generation routines for standard grids.
Options for incremental construction of grids.
Options for using a set of pre-defined stagger locations or for setting custom stagger locations.

31.1.1 Grid Representation in ESMF

ESMF Grids are based on the concepts described in *A Standard Description of Grids Used in Earth System Models* [Balaji 2006]. In this document Balaji introduces the mosaic concept as a means of describing a wide variety of Earth system model grids. A **mosaic** is composed of grid tiles connected at their edges. Mosaic grids includes simple, single tile grids as a special case.

The ESMF Grid class is a representation of a mosaic grid. Each ESMF Grid is constructed of one or more logically rectangular **Tiles**. A Tile will usually have some physical significance (e.g. the region of the world covered by one face of a cubed sphere grid).

The piece of a Tile that resides on one DE (for simple cases, a DE can be thought of as a processor - see section on the DELayout) is called a **LocalTile**. For example, the six faces of a cubed sphere grid are each Tiles, and each Tile can be divided into many LocalTiles.

Every ESMF Grid contains a DistGrid object, which defines the Grid's index space, topology, distribution, and connectivities. It enables the user to define the complex edge relationships of tripole and other grids. The DistGrid can be created explicitly and passed into a Grid creation routine, or it can be created implicitly if the user takes a Grid creation shortcut. The DistGrid used in Grid creation describes the properties of the Grid cells. In addition to this one, the Grid

internally creates DistGrids for each stagger location. These stagger DistGrids are related to the original DistGrid, but may contain extra padding to represent the extent of the index space of the stagger. These DistGrids are what are used when a Field is created on a Grid.

31.1.2 Supported Grids

The range of supported grids in ESMF can be defined by:

- Types of topologies and shapes supported. ESMF supports one or more logically rectangular grid Tiles with connectivities specified between cells. For more details see section 31.1.3.
- Types of distributions supported. ESMF supports regular, irregular, or arbitrary distributions of data. For more details see section 31.1.4.
- Types of coordinates supported. ESMF supports uniform, rectilinear, and curvilinear coordinates. For more details see section 31.1.5.

31.1.3 Grid Topologies and Periodicity

ESMF has shortcuts for the creation of standard Grid topologies or **shapes** up to 3D. In many cases, these enable the user to bypass the step of creating a DistGrid before creating the Grid. There are two sets of methods which allow the user to do this. These two sets of methods cover the same set of topologies, but allow the user to specify them in different ways.

The first set of these are a group of overloaded calls broken up by the number of periodic dimensions they specify. With these the user can pick the method which creates a Grid with the number of periodic dimensions they need, and then specify other connectivity options via arguments to the method. The following is a description of these methods:

ESMF_GridCreateNoPeriDim() Allows the user to create a Grid with no edge connections, for example, a regional Grid with closed boundaries.

ESMF_GridCreate1PeriDim() Allows the user to create a Grid with 1 periodic dimension and supports a range of options for what to do at the pole (see Section 31.2.5). Some examples of Grids which can be created here are tripole spheres, bipole spheres, cylinders with open poles.

ESMF_GridCreate2PeriDim() Allows the user to create a Grid with 2 periodic dimensions, for example a torus, or a regional Grid with doubly periodic boundaries.

More detailed information can be found in the API description of each.

The second set of shortcut methods is a set of methods overloaded under the name `ESMF_GridCreate()`. These methods allow the user to specify the connectivites at the end of each dimension, by using the ESMF_GridConn_Flag flag. The table below shows the ESMF_GridConn_Flag settings used to create standard shapes in 2D using the ESMF_GridCreate() call. Two values are specified for each dimension, one for the low end and one for the high end of the dimension's index values.

2D Shape	**connflagDim1(1)**	**connflagDim1(2)**	**connflagDim2(1)**	**connflagDim2(2)**
Rectangle	NONE	NONE	NONE	NONE
Bipole Sphere	POLE	POLE	PERIODIC	PERIODIC
Tripole Sphere	POLE	BIPOLE	PERIODIC	PERIODIC
Cylinder	NONE	NONE	PERIODIC	PERIODIC
Torus	PERIODIC	PERIODIC	PERIODIC	PERIODIC

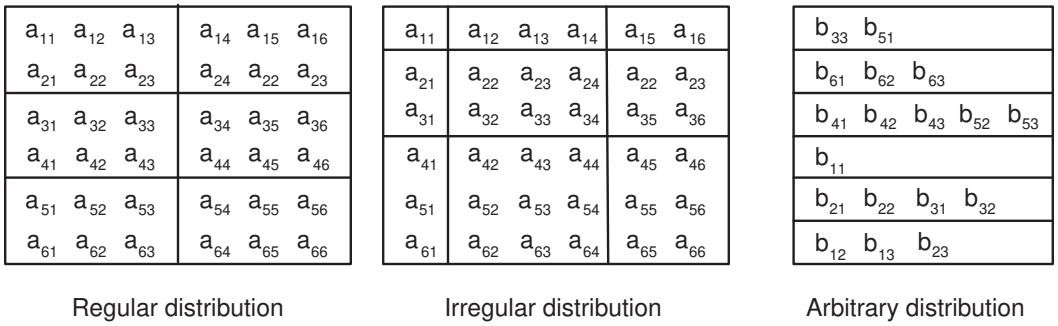


Figure 12: Examples of regular and irregular decomposition of a grid **a** that is 6x6, and an arbitrary decomposition of a grid **b** that is 6x3.

If the user's grid shape is too complex for an ESMF shortcut routine, or involves more than three dimensions, a DistGrid can be created to specify the shape in detail. This DistGrid is then passed into a Grid create call.

31.1.4 Grid Distribution

ESMF Grids have several options for data distribution (also referred to as decomposition). As ESMF Grids are cell based, these options are all specified in terms of how the cells in the Grid are broken up between DEs.

The main distribution options are regular, irregular, and arbitrary. A **regular** distribution is one in which the same number of contiguous grid cells are assigned to each DE in the distributed dimension. An **irregular** distribution is one in which unequal numbers of contiguous grid cells are assigned to each DE in the distributed dimension. An **arbitrary** distribution is one in which any grid cell can be assigned to any DE. Any of these distribution options can be applied to any of the grid shapes (i.e., rectangle) or types (i.e., rectilinear). Support for arbitrary distribution is limited in the current version of ESMF, see Section **??** for an example of creating a Grid with an arbitrary distribution.

Figure 12 illustrates options for distribution.

A distribution can also be specified using the DistGrid, by passing object into a Grid create call.

31.1.5 Grid Coordinates

Grid Tiles can have uniform, rectilinear, or curvilinear coordinates. The coordinates of **uniform** grids are equally spaced along their axes, and can be fully specified by the coordinates of the two opposing points that define the grid's physical span. The coordinates of **rectilinear** grids are unequally spaced along their axes, and can be fully specified by giving the spacing of grid points along each axis. The coordinates of **curvilinear grids** must be specified by giving the explicit set of coordinates for each grid point. Curvilinear grids are often uniform or rectilinear grids that have been warped; for example, to place a pole over a land mass so that it does not affect the computations performed on an ocean model grid. Figure 13 shows examples of each type of grid.

Each of these coordinate types can be set for each of the standard grid shapes described in section 31.1.3.

The table below shows how examples of common single Tile grids fall into this shape and coordinate taxonomy. Note that any of the grids in the table can have a regular or arbitrary distribution.

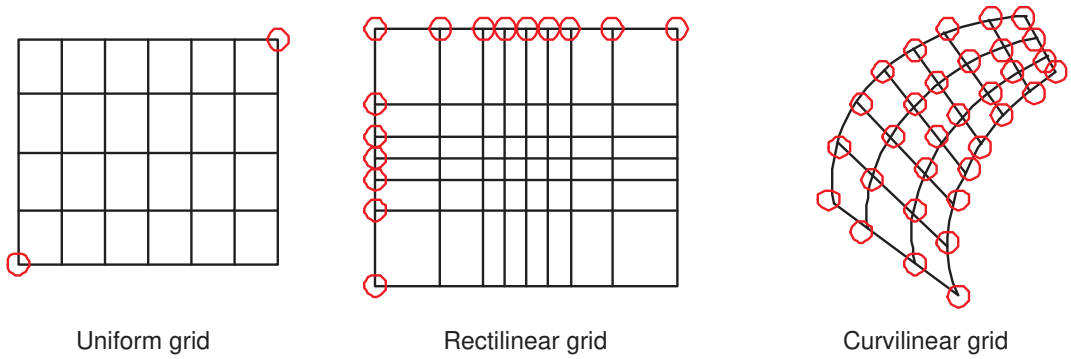


Figure 13: Types of logically rectangular grid tiles. Red circles show the values needed to specify grid coordinates for each type.

	Uniform	**Rectilinear**	**Curvilinear**
Sphere	Global uniform lat-lon grid	Gaussian grid	Displaced pole grid
Rectangle	Regional uniform lat-lon grid	Gaussian grid section	Polar stereographic grid section

31.1.6 Coordinate Specification and Generation

There are two ways of specifying coordinates in ESMF. The first way is for the user to **set** the coordinates. The second way is to take a shortcut and have the framework **generate** the coordinates.

See Section **??** for more description and examples of setting coordinates.

31.1.7 Staggering

Staggering is a finite difference technique in which the values of different physical quantities are placed at different locations within a grid cell.

The ESMF Grid class supports a variety of stagger locations, including cell centers, corners, and edge centers. The default stagger location in ESMF is the cell center, and cell counts in Grid are based on this assumption. Combinations of the 2D ESMF stagger locations are sufficient to specify any of the Arakawa staggers. ESMF also supports staggering in 3D and higher dimensions. There are shortcuts for standard staggers, and interfaces through which users can create custom staggers.

As a default the ESMF Grid class provides symmetric staggering, so that cell centers are enclosed by cell perimeter (e.g. corner) stagger locations. This means the coordinate arrays for stagger locations other than the center will have an additional element of padding in order to enclose the cell center locations. However, to achieve other types of staggering, the user may alter or eliminate this padding by using the appropriate options when adding coordinates to a Grid.

In the current release, only the cell center stagger location is supported for an arbitrarily distributed grid. For examples and a full description of the stagger interface see Section **??**.

31.1.8 Masking

Masking is the process whereby parts of a Grid can be marked to be ignored during an operation. For a description of how to set mask information in the Grid, see here **??**. For a description of how masking works in regridding, see here 24.2.10.

31.2 Constants

31.2.1 ESMF_GRIDCONN

DESCRIPTION:

The `ESMF_GridCreateShapeTile` command has three specific arguments `connflagDim1`, `connflagDim2`, and `connflagDim3`. These can be used to setup different types of connections at the ends of each dimension of a Tile. Each of these parameters is a two element array. The first element is the connection type at the minimum end of the dimension and the second is the connection type at the maximum end. The default value for all the connections is ESMF_GRIDCONN_NONE, specifying no connection.

The type of this flag is:

`type(ESMF_GridConn_Flag)`

The valid values are:

ESMF_GRIDCONN_NONE No connection.

ESMF_GRIDCONN_PERIODIC Periodic connection.

ESMF_GRIDCONN_POLE This edge is connected to itself. Given that the edge is n elements long, then element i is connected to element ((i+n/2) mod n).

ESMF_GRIDCONN_BIPOLE This edge is connected to itself. Given that the edge is n elements long, element i is connected to element n-i+1.

31.2.2 ESMF_GRIDITEM

DESCRIPTION:
The ESMF Grid can contain other kinds of data besides coordinates. This data is referred to as Grid "items". Some items may be used by ESMF for calculations involving the Grid. The following are the valid values of ESMF_GridItem_Flag.

The type of this flag is:

`type(ESMF_GridItem_Flag)`

The valid values are:

Item Label	Type Restriction	Type Default	ESMF Uses	Controls
ESMF_GRIDITEM_MASK	ESMF_TYPEKIND_I4	ESMF_TYPEKIND_I4	YES	Masking in Regrid
ESMF_GRIDITEM_AREA	NONE	ESMF_TYPEKIND_R8	YES	Conservation in Regrid

NOTE: One important thing to consider when setting areas in the Grid using `ESMF_GRIDITEM_AREA`, ESMF doesn't currently do unit conversion on areas. If these areas are going to be used in a process that also involves the

areas of another Grid or Mesh (e.g. conservative regridding), then it is the user's responsibility to make sure that the area units are consistent between the two sides. If ESMF calculates an area on the surface of a sphere, then it is in units of square radians. If it calculates the area for a Cartesian grid, then it is in the same units as the coordinates, but squared.

31.2.3 ESMF_GRIDMATCH

DESCRIPTION:
This type is used to indicate the level to which two grids match.

The type of this flag is:

`type(ESMF_GridMatch_Flag)`

The valid values are:

ESMF_GRIDMATCH_INVALID: Indicates a non-valid matching level. Returned if an error occurs in the matching function. If a higher matching level is returned then no error occurred.

ESMF_GRIDMATCH_NONE: The lowest level of grid matching. This indicates that the Grid's don't match at any of the higher levels.

ESMF_GRIDMATCH_EXACT: All the pieces of the Grid (e.g. distgrids, coordinates, etc.) except the name, match between the two Grids.

ESMF_GRIDMATCH_ALIAS: Both Grid variables are aliases to the exact same Grid object in memory.

31.2.4 ESMF_GRIDSTATUS

DESCRIPTION:
The ESMF Grid class can exist in two states. These states are present so that the library code can detect if a Grid has been appropriately setup for the task at hand. The following are the valid values of ESMF_GRIDSTATUS.

The type of this flag is:

`type(ESMF_GridStatus_Flag)`

The valid values are:

ESMF_GRIDSTATUS_EMPTY: Status after a Grid has been created with `ESMF_GridEmptyCreate`. A Grid object container is allocated but space for internal objects is not. Topology information and coordinate information is incomplete. This object can be used in `ESMF_GridEmptyComplete()` methods in which additional information is added to the Grid.

ESMF_GRIDSTATUS_COMPLETE: The Grid has a specific topology and distribution, but incomplete coordinate arrays. The Grid can be used as the basis for allocating a Field, and coordinates can be added via `ESMF_GridCoordAdd()` to allow other functionality.

31.2.5 ESMF_POLEKIND

DESCRIPTION:
This type describes the type of connection that occurs at the pole when a Grid is created with `ESMF_GridCreate1PeriodicDim()`.

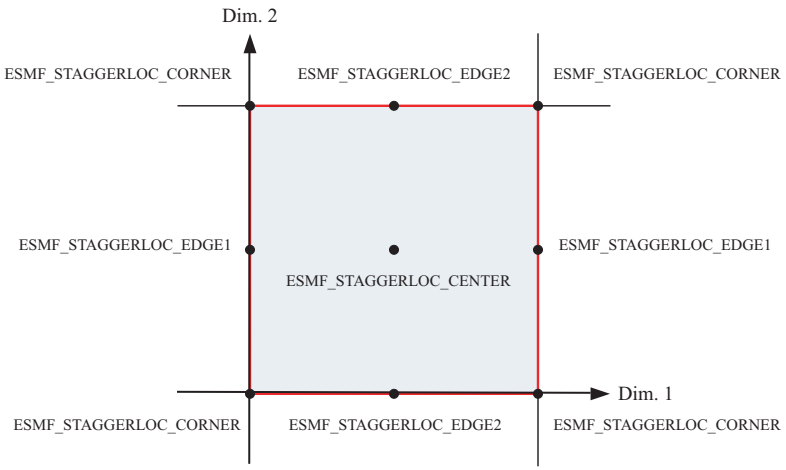


Figure 14: 2D Predefined Stagger Locations

The type of this flag is:

`type(ESMF_PoleKind_Flag)`

The valid values are:

ESMF_POLEKIND_NONE No connection at pole.

ESMF_POLEKIND_MONOPOLE This edge is connected to itself. Given that the edge is n elements long, then element i is connected to element i+n/2.

ESMF_POLEKIND_BIPOLE This edge is connected to itself. Given that the edge is n elements long, element i is connected to element n-i+1.

31.2.6 ESMF_STAGGERLOC

DESCRIPTION:
In the ESMF Grid class, data can be located at different positions in a Grid cell. When setting or retrieving coordinate data the stagger location is specified to tell the Grid method from where in the cell to get the data. Although the user may define their own custom stagger locations, ESMF provides a set of predefined locations for ease of use. The following are the valid predefined stagger locations.

The 2D predefined stagger locations (illustrated in figure 14) are:

ESMF_STAGGERLOC_CENTER: The center of the cell.

ESMF_STAGGERLOC_CORNER: The corners of the cell.

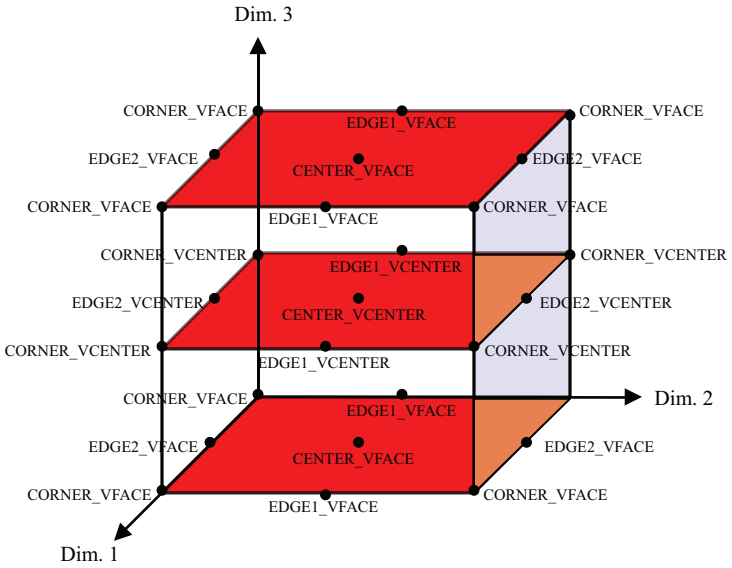


Figure 15: 3D Predefined Stagger Locations

ESMF_STAGGERLOC_EDGE1: The edges offset from the center in the 1st dimension.

ESMF_STAGGERLOC_EDGE2: The edges offset from the center in the 2nd dimension.

The 3D predefined stagger locations (illustrated in figure 15) are:

ESMF_STAGGERLOC_CENTER_VCENTER: The center of the 3D cell.

ESMF_STAGGERLOC_CORNER_VCENTER: Half way up the vertical edges of the cell.

ESMF_STAGGERLOC_EDGE1_VCENTER: The center of the face bounded by edge 1 and the vertical dimension.

ESMF_STAGGERLOC_EDGE2_VCENTER: The center of the face bounded by edge 2 and the vertical dimension.

ESMF_STAGGERLOC_CORNER_VFACE: The corners of the 3D cell.

ESMF_STAGGERLOC_EDGE1_VFACE: The center of the edges of the 3D cell parallel offset from the center in the 1st dimension.

ESMF_STAGGERLOC_EDGE2_VFACE: The center of the edges of the 3D cell parallel offset from the center in the 2nd dimension.

ESMF_STAGGERLOC_CENTER_VFACE: The center of the top and bottom face. The face bounded by the 1st and 2nd dimensions.

31.3 Use and Examples

This section describes the use of the ESMF Grid class. It first discusses the more user friendly shape specific interface to the Grid. During this discussion it covers creation and options, adding stagger locations, coordinate data access, and other grid functionality. After this initial phase the document discusses the more advanced options which the user can employ should they need more customized interaction with the Grid class.

31.4 Restrictions and Future Work

- **Grids with factorized coordinates can only be redisted when they are 2D.** Using the ESMF_GridCreate() interface that allows the user to create a copy of an existing Grid with a new distribution will give incorrect results when used on a Grid with 3 or more dimensions and whose coordinate arrays are less than the full dimension of the Grid (i.e. it contains factorized coordinates).
- **7D limit.** Only grids up to 7D will be supported.
- **Future adaptation.** Currently Grids are created and then remain unchanged. In the future, it would be useful to provide support for the various forms of grid adaptation. This would allow the grids to dynamically change their resolution to more closely match what is needed at a particular time and position during a computation for front tracking or adaptive meshes.
- **Future Grid generation.** This class for now only contains the basic functionality for operating on the grid. In the future methods will be added to enable the automatic generation of various types of grids.

31.5 Design and Implementation Notes

31.5.1 Grid Topology

The `ESMF_Grid` class depends upon the `ESMF_DistGrid` class for the specification of its topology. That is, when creating a Grid, first an `ESMF_DistGrid` is created to describe the appropriate index space topology. This decision was made because it seemed redundant to have a system for doing this in both classes. It also seems most appropriate for the machinary for topology creation to be located at the lowest level possible so that it can be used by other classes (e.g. the `ESMF_Array` class). Because of this, however, the authors recommend that as a natural part of the implementation of subroutines to generate standard grid shapes (e.g. `ESMF_GridGenSphere`) a set of standard topology generation subroutines be implemented (e.g. `ESMF_DistGridGenSphere`) for users who want to create a standard topology, but a custom geometry.

31.6 Class API: General Grid Methods

31.7 Class API: StaggerLoc Methods

32 LocStream Class

32.1 Description

A location stream (LocStream) can be used to represent the locations of a set of data points. For example, in the data assimilation world, LocStreams can be used to represent a set of observations. The values of the data points are stored within a Field or FieldBundle created using the LocStream.

The locations are generally described using Cartesian (x, y, z), or (lat, lon, radius) coordinates. The coordinates are stored using constructs called *keys*. A Key is essentially a list of point descriptors, one for each data point. They may hold other information besides the coordinates - a mask, for example. They may also hold a second set of coordinates. Keys are referenced by name - see 32.2.1 and 32.2.2 for specific keynames required in regridding. Each key must contain the same number of elements as there are data points in the LocStream. While there is no assumption in the ordering of the points, the order chosen must be maintained in each of the keys.

LocStreams can be very large. Data assimilation systems might use LocStreams with up to 10^8 observations, so efficiency is critical. LocStreams can be created from file, see **??**.

Common operations involving LocStreams are similar to those involving Grids. For example, LocStreams allow users to:

1. Create a Field or FieldBundle on a LocStream
2. Regrid data in Fields defined on LocStreams
3. Redistribute data between Fields defined on LocStreams
4. Gather or scatter a FieldBundle defined on a LocStream from/to a root DE
5. Extract Fortran array from Field which was defined on a LocStream

A LocStream differs from a Grid in that no topological structure is maintained between the points (e.g. the class contains no information about which point is the neighbor of which other point).

A LocStream is similar to a Mesh in that both are collections of irregularly positioned points. However, the two structures differ because a Mesh also has connectivity: each data point represents either a center or corner of a cell. There is no requirement that the points in a LocStream have connectivity, in fact there is no requirement that any two points have any particular spatial relationship at all.

32.2 Constants

32.2.1 Coordinate keyNames

DESCRIPTION:
For ESMF to be able to use coordinates specified in a LocStream key (e.g. in regridding) they need to be named with the appropriate identifiers. The particular identifiers depend on the coordinate system (i.e. coordSys argument) used to create the LocStream containing the keys. ESMF regridding expects these keys to be of type ESMF_TYPEKIND_R8.

The valid values are:

Coordinate System	**dimension 1**	**dimension 2**	**dimension 3 (if used)**
ESMF_COORDSYS_SPH_DEG	ESMF:Lon	ESMF:Lat	ESMF:Radius
ESMF_COORDSYS_SPH_RAD	ESMF:Lon	ESMF:Lat	ESMF:Radius
ESMF_COORDSYS_CART	ESMF:X	ESMF:Y	ESMF:Z

32.2.2 Masking keyName

DESCRIPTION:
Points within a LocStream can be marked and then potentially ignored during certain operations, like regridding. This masking information must be contained in a key named with the appropriate identifier. ESMF regridding expects this key to be of type ESMF_TYPEKIND_I4.

The valid value is:

ESMF:Mask

32.3 Use and Examples

32.4 Class API

33 Mesh Class

33.1 Description

Unstructured grids are commonly used in the computational solution of partial differential equations. These are especially useful for problems that involve complex geometry, where using the less flexible structured grids can result in grid representation of regions where no computation is needed. Finite element and finite volume methods map naturally to unstructured grids and are used commonly in hydrology, ocean modeling, and many other applications.

In order to provide support for application codes using unstructured grids, the ESMF library provides a class for representing unstructured grids called the **Mesh**. Fields can be created on a Mesh to hold data. Fields created on a Mesh can also be used as either the source or destination or both of an interpolation (i.e. an `ESMF_FieldRegridStore()` call) which allows data to be moved between unstructured grids. This section describes the Mesh and how to create and use them in ESMF.

33.1.1 Mesh representation in ESMF

A Mesh in ESMF is constructed of **nodes** and **elements**.

A **node**, also known as a vertex or corner, is a part of a Mesh which represents a single point. Coordinate information is set in a node.

An **element**, also known as a cell, is a part of a mesh which represents a small region of space. Elements are described in terms of a connected set of nodes which represent locations along their boundaries.

Field data may be located on either the nodes or elements of a Mesh.

The dimension of a Mesh in ESMF is specified with two parameters: the **parametric dimension** and the **spatial dimension**.

The **parametric dimension** of a Mesh is the dimension of the topology of the Mesh. This can be thought of as the dimension of the elements which make up the Mesh. For example, a Mesh composed of triangles would have a parametric dimension of 2, whereas a Mesh composed of tetrahedra would have a parametric dimension of 3.

The **spatial dimension** of a Mesh is the dimension of the space the Mesh is embedded in. In other words, it is the number of coordinate dimensions needed to describe the location of the nodes making up the Mesh.

For example, a Mesh constructed of squares on a plane would have a parametric dimension of 2 and a spatial dimension of 2. If that same Mesh were used to represent the 2D surface of a sphere, then the Mesh would still have a parametric dimension of 2, but now its spatial dimension would be 3.

33.1.2 Supported Meshes

The range of Meshes supported by ESMF are defined by several factors: dimension, element types, and distribution.

ESMF currently only supports Meshes whose number of coordinate dimensions (spatial dimension) is 2 or 3. The dimension of the elements in a Mesh (parametric dimension) must be less than or equal to the spatial dimension, but also must be either 2 or 3. This means that a Mesh may be either 2D elements in 2D space, 3D elements in 3D space, or a manifold constructed of 2D elements embedded in 3D space.

ESMF supports a range of elements for each Mesh parametric dimension. For a parametric dimension of 2, the native supported element types are triangles and quadrilaterals. In addition to these, ESMF supports 2D polygons with any number of sides. Internally these are represented as sets of triangles, but to the user should behave like any other element. For a parametric dimension of 3, the supported element types are tetrahedrons and hexahedrons. See Section 33.2.1 for diagrams of these. The Mesh supports any combination of element types within a particular dimension, but types from different dimensions may not be mixed. For example, a Mesh cannot be constructed of both quadrilaterals and tetrahedra.

ESMF currently only supports distributions where every node on a PET must be a part of an element on that PET. In other words, there must not be nodes without a corresponding element on any PET.

33.2 Constants

33.2.1 ESMF_MESHELEMTYPE

DESCRIPTION:
An ESMF Mesh can be constructed from a combination of different elements. The type of elements that can be used in a Mesh depends on the Mesh's parameteric dimension, which is set during Mesh creation. The following are the valid Mesh element types for each valid Mesh parametric dimension (2D or 3D) .

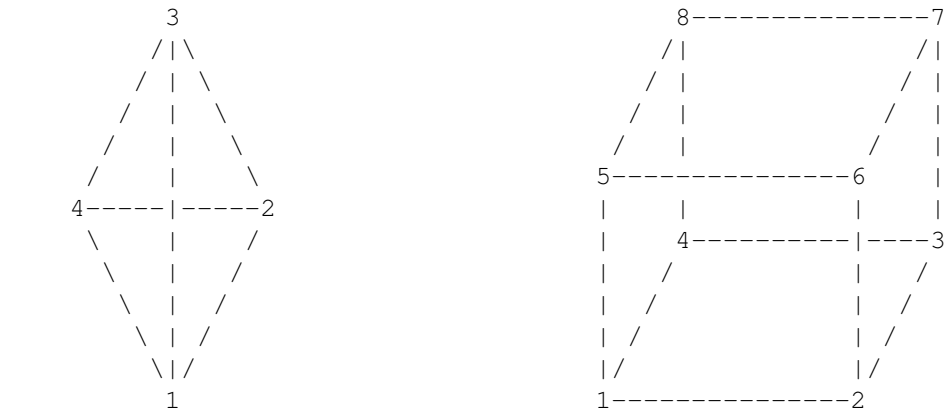
ESMF_MESHELEMTYPE_TRI ESMF_MESHELEMTYPE_QUAD

2D element types (numbers are the order for elementConn during Mesh create)

For a Mesh with parametric dimension of 2 ESMF supports two native element types (illustrated above), but also supports polygons with more sides. Internally these polygons are represented as a set of triangles, but to the user

should behave like other elements. To specify the non-native polygons in the `elementType` argument use the number of corners of the polygon (e.g. for a pentagon use 5). The connectivity for a polygon should be specified in counterclockwise order. The following table summarizes this information:

Element Type	Number of Nodes	Description
ESMF_MESHELEMTYPE_TRI	3	A triangle
ESMF_MESHELEMTYPE_QUAD	4	A quadrilateral (e.g. a rectangle)
N	N	An N-gon (e.g. if N=5 a pentagon)



ESMF_MESHELEMTYPE_TETRA ESMF_MESHELEMTYPE_HEX

3D element types (numbers are the order for elementConn during Mesh create)

For a Mesh with parametric dimension of 3 the valid element types (illustrated above) are:

Element Type	Number of Nodes	Description
ESMF_MESHELEMTYPE_TETRA	4	A tetrahedron (NOT VALID IN BILINEAR OR PATCH REGRID)
ESMF_MESHELEMTYPE_HEX	8	A hexahedron (e.g. a cube)

33.3 Use and Examples

33.4 Class API

34 XGrid Class

34.1 Description

An exchange grid represents the 2D boundary layer usually between the atmosphere on one side and ocean and land on the other in an Earth system model. There are dynamical and thermodynamical processes on either side of the boundary layer and on the boundary layer itself. The boundary layer exchanges fluxes from either side and adjusts boundary conditions for the model components involved. For climate modeling, it is critical that the fluxes transferred by the boundary layer are conservative.

The ESMF exchange grid is implemented as the ESMF_XGrid class. Internally it's represented by a collection of the intersected cells between atmosphere and ocean/land[10] grids. These polygonal cells can have irregular shapes and can be broken down into triangles facilitating a finite element approach.

There are two ways to create an ESMF_XGrid object from user supplied information. The first way to create an ESMF_XGrid takes two lists of ESMF_Grid or ESMF_Mesh that represent the model component grids on either side of the exchange grid. From the two lists of ESMF_Grid or ESMF_Mesh, information required for flux exchange calculation between any pair of the model components from either side of the exchange grid is computed. In addition, the internal representation of the ESMF_XGrid is computed and can be optionally stored as an ESMF_Mesh. This internal representation is the collection of the intersected polygonal cells as a result of merged ESMF_Meshes from both sides of the exchange grid. ESMF_Field can be created on the ESMF_XGrid and used for weight generation and regridding as the internal representation in the ESMF_XGrid has a complete geometrical description of the exchange grid.

The second way to create an ESMF_XGrid requires users to supply all necessary information to compute communication routehandle. A later call to ESMF_FieldRegridStore() with the xgrid and source and destination ESMF_Fields computes the ESMF_Routehandle object for matrix multiply operation used in model remapping.

ESMF_XGrid deals with 2 distinctive kinds of fraction for each Grid or Mesh cell involved in its creation. The following description applies to both ESMF_Grid and ESMF_Mesh involved in the ESMF_XGrid creation process. The first fraction quantity f_1 is the same as defined in direct Field regrid between a source and destination ESMF_Field pair, namely the fraction of a total Grid cell area A that is used in weight generation. The second fraction quantity f_2 is a result of the Grid merging process when multiple ESMF_Grids or model components exist on one side of the exchange grid. To compute XGrid, the multiple ESMF_Grids are first merged together to form a super mesh. During the merging process, Grids that are of a higher priority clips into lower priority Grids, creating fractional cells in the lower priority Grids. Priority is a mechanism to resolve the claim of a surface region by multiple Grids. To conserve flux, any surface area can only be claimed by a unique Grid. This is a typical practice in earth system modelling, e.g. to handle land and ocean boundary.

In addition to the matrix multiply communication routehandle, ESMF_XGrid exports both f_1 and f_2 to the user through the ESMF_FieldRegridStore() method because each remapping pair has different f_1 and f_2 associated with it. f_2 from source Grid is folded directly in the calculated weight matrices since its used to calculate destination point flux density F. The global source flux is defined as $\sum_{g=1}^{g=n\_srcgrid} \sum_{s=1}^{s=n\_srccell} f_{1s} f_{2s} A_s F_s$. The global destination flux is defined as: $\sum_{g=1}^{g=n\_dstgrid} \sum_{d=1}^{d=n\_dstcell} \sum_{s=1}^{s=n\_intersect} (w_{sd} F_s) f_{2d} A_d$, w_{sd} is the f_2 modified weight intersecting s-th source Grid cell with d-th destination Grid cell. It can be proved that this formulation of the fractions and weight calculation ensures first order global conservation of flux $\mathcal{F}$ transferred from source grids to exchange grid, and from exchange grid to destination grids.

34.2 Constants

34.2.1 ESMF_XGRIDSIDE

DESCRIPTION:
Specify which side of the ESMF_XGrid the current operation is taking place.

The type of this flag is:

type(ESMF_XGridSide_Flag)

The valid values are:

ESMF_XGRIDSIDE_A A side of the eXchange Grid, corresponding to the A side of the Grids used to create an XGrid.

ESMF_XGRIDSIDE_B B side of the eXchange Grid, corresponding to the B side of the Grids used to create an XGrid.

ESMF_XGRIDSIDE_BALANCED The internally generated balanced side of the eXchange Grid in the middle.

34.3 Use and Examples

34.4 Restrictions and Future Work

34.4.1 Restrictions and Future Work

1. **CAUTION:** Any Grid or Mesh pair picked from the A side and B side of the XGrid cannot point to the same Grid or Mesh in memory on a local PET. This prevents Regrid from selecting the right source and destination grid automatically to calculate the regridding routehandle. It's okay for the Grid and Mesh to have identical topological and geographical properties as long as they are stored in different memory.

34.5 Design and Implementation Notes

1. The XGrid class is implemented in Fortran, and as such is defined inside the framework by a XGrid derived type and a set of subprograms (functions and subroutines) which operate on that derived type. The XGrid class contains information needed to create Grid, Field, and communication routehandle.

2. XGrid follows the framework-wide convention of the *unison* creation and operation rule: All PETs which are part of the currently executing VM must create the same XGrids at the same point in their execution. In addition to the unison rule, XGrid creation also performs inter-PET communication within the current executing VM.

34.6 Class API

35 Geom Class

35.1 Description

The ESMF Geom class is used as a container for other ESMF geometry objects (e.g. an ESMF Grid). This allows a generic representation of a geometry to be passed around (e.g. through a coupled system) without it's specific type being known. Some operations are supported on a Geom object and more will be added over time as needed. However, if an unsupported operation is needed, then the specific geometry object can always be pulled out and operated on that way.

In addition to the geometry object, a Geom can also contain information describing a location on a geometry. For example, in the case of a Grid, a geometry object will also contain a stagger location. Having this location information allows the creation of Fields and other capabilities to be performed in the most generic way on a Geom object. For geometries where it is appropriate, the user can optionally specify this location information during the creation of a Geom object. However, if no location is specified, then default values for this information are provided which match those which would be used when creating a Field with the specific geometry (e.g. stagger location ESMF_STAGGERLOC_CENTER for a Grid).

35.2 Class API: Geom Methods

36 DistGrid Class

36.1 Description

The ESMF DistGrid class sits on top of the DELayout class and holds domain information in index space. A DistGrid object captures the index space topology and describes its decomposition in terms of DEs. Combined with DELayout and VM the DistGrid defines the data distribution of a domain decomposition across the computational resources of an ESMF Component.

The global domain is defined as the union of logically rectangular (LR) sub-domains or *tiles*. The DistGrid create methods allow the specification of such a multi-tile global domain and its decomposition into exclusive, DE-local LR regions according to various degrees of user specified constraints. Complex index space topologies can be constructed by specifying connection relationships between tiles during creation.

The DistGrid class holds domain information for all DEs. Each DE is associated with a local LR region. No overlap of the regions is allowed. The DistGrid offers query methods that allow DE-local topology information to be extracted, e.g. for the construction of halos by higher classes.

A DistGrid object only contains decomposable dimensions. The minimum rank for a DistGrid object is 1. A maximum rank does not exist for DistGrid objects, however, ranks greater than 7 may lead to difficulties with respect to the Fortran API of higher classes based on DistGrid. The rank of a DELayout object contained within a DistGrid object must be equal to the DistGrid rank. Higher class objects that use the DistGrid, such as an Array object, may be of different rank than the associated DistGrid object. The higher class object will hold the mapping information between its dimensions and the DistGrid dimensions.

36.2 Constants

36.2.1 ESMF_DISTGRIDMATCH

DESCRIPTION:
Indicates the level to which two DistGrid variables match.

The type of this flag is:

`type(ESMF_DistGridMatch_Flag)`

The valid values are:

ESMF_DISTGRIDMATCH_INVALID: Indicates a non-valid matching level. One or both DistGrid objects are invalid.

ESMF_DISTGRIDMATCH_NONE: The lowest valid level of DistGrid matching. This indicates that the DistGrid objects don't match at any of the higher levels.

ESMF_DISTGRIDMATCH_INDEXSPACE: The index space covered by the two DistGrid objects is identical. However, differences between the two objects prevents a higher matching level.

ESMF_DISTGRIDMATCH_TOPOLOGY: The topology (i.e. index space and connections) defined by the two DistGrid objects is identical. However, differences between the two objects prevents a higher matching level.

ESMF_DISTGRIDMATCH_DECOMP: The index space decomposition defined by the two DistGrid objects is identical. However, differences between the two objects prevents a higher matching level.

ESMF_DISTGRIDMATCH_EXACT: The two DistGrid objects match in all aspects, including sequence indices. The only aspect that may differ between the two objects is their name.

ESMF_DISTGRIDMATCH_ALIAS: Both DistGrid variables are aliases to the exact same DistGrid object in memory.

36.3 Use and Examples

The following examples demonstrate how to create, use and destroy DistGrid objects. In order to produce complete and valid DistGrid objects all of the `ESMF_DistGridCreate()` calls require to be called in unison i.e. on *all* PETs of a component with a complete set of valid arguments.

36.4 Restrictions and Future Work

- Multi-tile DistGrids from deBlockList are not yet supported.
- The fastAxis feature has not been implemented yet.

36.5 Design and Implementation Notes

This section will be updated as the implementation of the DistGrid class nears completion.

36.6 Class API

36.7 Class API: DistGridConnection Methods

36.8 Class API: DistGridRegDecomp Methods

37 RouteHandle Class

37.1 Description

The ESMF RouteHandle class provides a unified interface for all route-based communication methods across the Field, FieldBundle, Array, and ArrayBundle classes. All route-based communication methods implement a pre-computation step, returning a RouteHandle, an execution step, and a release step. Typically the pre-computation, or Store() step will be a lot more expensive (both in memory and time) than the execution step. The idea is that once precomputed, a RouteHandle will be executed many times over during a model run, making the execution time a very performance critical piece of code. In ESMF, Regridding, Redisting, and Haloing are implemented as route-based communication methods. The following sections discuss the RouteHandle concepts that apply uniformly to all route-based communication methods, across all of the above mentioned classes.

37.2 Use and Examples

The user interacts with the RouteHandle class through the route-based communication methods of Field, FieldBundle, Array, and ArrayBundle. The usage of these methods are described in detail under their respective class documentation section. The following examples focus on the RouteHandle aspects common across classes and methods.

37.3 Restrictions and Future Work

- **Non-blocking** communication via the `routesyncflag` option is implemented for Fields and Arrays. It is *not* available for FieldBundles and ArrayBundles. The user is advised to use the VMEpoch approach for all cases to achive asynchronicity.
- The **dynamic masking** feature currently has the following limitations:
  - Only available for `ESMF_TYPEKIND_R8` and `ESMF_TYPEKIND_R4` Fields and Arrays.
  - Only available through the `ESMF_FieldRegrid()` and `ESMF_ArraySMM()` methods.
  - Destination objects that have undistributed dimensions *after* any distributed dimension are not supported.
  - No check is implemented that ensure the user-provided RouteHandle object is suitable for dynamic masking.

37.4 Design and Implementation Notes

Internally all route-based communication calls are implemented as sparse matrix multiplications. The precompute step for all of the supported communication methods can be broken up into three steps:

1. Construction of the sparse matrix for the specific communication method.
2. Generation of the communication pattern according to the sparse matrix.
3. Encoding of the communication pattern for each participating PET in form of an XXE stream.

37.5 Class API

38 I/O Capability

38.1 Description

The ESMF I/O provides a unified interface for input and output of high level ESMF objects such as Fields. ESMF I/O capability is integrated with third-party software such as Parallel I/O (PIO) to read and write Fortran array data in NetCDF format, and JSON for Modern C++ Library to read Info attribute data in JSON format. Other file I/O functionalities, such as writing of error and log messages, input of configuration parameters from an ASCII file, and lower-level I/O utilities are covered in different sections of this document. See the Log Class 49.1, the Config Class 47.1, and the Fortran I/O Utilities, 53.1 respectively.

38.2 Data I/O

ESMF provides interfaces for high performance, parallel I/O using ESMF data objects such as Arrays and Fields. Currently ESMF only supports I/O of NetCDF files. The current ESMF implementation relies on the Parallel I/O (PIO) library developed as a collaboration between NCAR and DOE laboratories. PIO is built as part of the ESMF build when the environment variable ESMF_PIO is set to "internal", or is linked against when ESMF_PIO is set to "external"; by default ESMF_PIO is not set (which results in using the internal PIO if other aspects of the ESMF build configuration allow it). When PIO is built with ESMF, the ESMF methods internally call the PIO interfaces. When ESMF is not built with PIO, the ESMF methods are non-operable (no-op) stubs that simply return with a return code of ESMF_RC_LIB_NOT_PRESENT. Details about the environment variables can be found in ESMF User Guide, "Building and Installing the ESMF", "Third Party Libraries".

The following methods support parallel data I/O using PIO:

`ESMF_FieldBundleRead()`, section **??**.

`ESMF_FieldBundleWrite()`, section **??**.

`ESMF_FieldRead()`, section **??**.

`ESMF_FieldWrite()`, section **??**.

`ESMF_ArrayBundleRead()`, section **??**.

`ESMF_ArrayBundleWrite()`, section **??**.

`ESMF_ArrayRead()`, section **??**.

`ESMF_ArrayWrite()`, section **??**.

38.3 Data formats

The only format currently supported is NetCDF. The environment variables ESMF_NETCDF and/or ESMF_PNETCDF must be set to enable this NetCDF-based I/O. Details about the environment variables can be found in ESMF User Guide, "Building and Installing the ESMF", "Third Party Libraries".

NetCDF Network Common Data Form (NetCDF) is an interface for array-oriented data access. The NetCDF library provides an implementation of the interface. It also defines a machine-independent format for representing scientific data. Together, the interface, library, and format support the creation, access, and sharing of scientific

data. The NetCDF software was developed at the Unidata Program Center in Boulder, Colorado. See [9]. In geoscience, NetCDF can be naturally used to represent fields defined on logically rectangular grids. NetCDF use in geosciences is specified by CF conventions mentioned above [8].

To the extent that data on unstructured grids (or even observations) can be represented as one-dimensional arrays, NetCDF can also be used to store these data. However, it does not provide a high-level abstraction for this type of data.

38.4 Restrictions and Future Work

1. **Limited data formats supported.** Currently a small fraction of the anticipated data formats is implemented by ESMF. The data I/O uses NetCDF format, and ESMF Info I/O uses JSON format. Different libraries are employed for these different formats. In future development, a more centralized I/O technique will likely be defined to provide efficient utilities with a set of standard APIs that will allow manipulation of multiple standard formats. Also, the ability to automatically detect file formats at runtime will be developed.

2. **Some limitations with multi-tile I/O.** There are a few limitations when doing I/O on multi-tile Arrays and Fields (e.g., a cubed sphere grid represented as a six-tile grid): This I/O requires at least as many PETs as there are tiles, and for I/O of ArrayBundles and FieldBundles, all Arrays / Fields in the bundle must contain the same number of tiles.

3. **Replicated dimensions.** I/O of Arrays / Fields with replicated dimensions (section **??**) is only partially working. In most situations, replicated dimensions appear as dimensions in the output file; ideally, these replicated dimensions would be removed in the output file, and we plan to make that change in the future. Furthermore, slices of the replicated dimensions other than the first can have garbage values in the output file. In addition, there is an inconsistency when outputting Arrays / Fields that have a decomposition with more than one DE per PET: in this case, replicated dimensions are removed in the output file. Finally, I/O cannot be performed on multi-tile Arrays / Fields with replicated dimensions.

38.5 Design and Implementation Notes

For data I/O, the ESMF I/O capability relies on the PIO and NetCDF libraries, and optionally the PNetCDF library. For Info attribute I/O, the ESMF I/O capability uses the JSON for Modern C++ library to perform reading of JSON files. PIO and JSON for Modern C++ are included with the ESMF distribution; the other libraries must be installed on the machine of interest.

Part V

Infrastructure: Utilities

39 Overview of Infrastructure Utility Classes

The ESMF utilities are a set of tools for quickly assembling modeling applications.

The ESMF Info class enables models to be self-describing via metadata, which are instances of JSON-compatible key-value pairs.

The Time Management Library provides utilities for time and time interval representation and calculation, and higher-level utilities that control model time stepping, via clocks, as well as alarming.

The ESMF Config class provides configuration management based on NASA DAO's Inpak package, a collection of methods for accessing files containing input parameters stored in an ASCII format.

The ESMF LogErr class consists of a variety of methods for writing error, warning, and informational messages to log files. A default Log is created during ESMF initialization. Other Logs can be created later in the code by the user.

The DELayout class provides a layer of abstraction on top of the Virtual Machine (VM) layer. DELayout does this by introducing DEs (Decomposition Elements) as logical resource units. The DELayout object keeps track of the relationship between its DEs and the resources of the associated VM object. A DELayout can be shaped by the user at creation time to best match the computational problem or other design criteria.

The ESMF VM (Virtual Machine) class is a generic representation of hardware and system software resources. There is exactly one VM object per ESMF Component, providing the execution environment for the Component code. The VM class handles all resource management tasks for the Component class and provides a description of the underlying configuration of the compute resources used by a Component. In addition to resource description and management, the VM class offers the lowest level of ESMF communication methods.

The ESMF Fortran I/O utilities provide portable methods to access capabilities which are often implemented in different ways amongst different environments. Currently, two utility methods are implemented: one to find an unopened unit number, and one to flush an I/O buffer.

40 Info Class (Object Attributes)

All ESMF base objects (i.e. Array, ArrayBundle, Field, FieldBundle, Grid, Mesh, DistGrid) contain a key-value attribute storage object called ESMF_Info. ESMF_Info objects may also be created independent of a base object. ESMF_Info supports setting and getting key-value pairs where the *key* is a string and the *value* is a scalar or a list of common data types. An ESMF_Info object may have a flat or nested data structure. The purpose of ESMF_Info is to support I/O-compatible metadata structures (i.e. netCDF), internal record-keeping for model execution (NUOPC), and provide a mechanism for custom user metadata attributes.

ESMF_Info is designed for interoperability. To achieve this goal, ESMF_Info adopted the JSON (Javascript Object Notation) specification. Internally, ESMF_Info uses *JSON for Modern C++* [1] to manage its storage map. There are numerous resources for JSON on the web [6]. Quoting from the *json.org* site [6] when it introduces the format:

> *JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language. JSON is built on two structures:*
>
> - *A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.*
> - *An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.*
>
> *These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.*

By adopting JSON compliance for ESMF_Info, ESMF made its core metadata capabilities explicitly interoperable with a widely used data structure. If data may be represented with JSON, then it is compatible with ESMF_Info.

There are some aspects of the ESMF_Info implementation related to JSON and JSON for Modern C++ that should be noted:

1. JSON supports 64-bit data types for integers and reals ([3], [2]). I4/R4 is converted to I8/R8 and vice versa. ESMF_Info internally tracks 32-bit sets to ensure the data type may be appropriately queried.
2. The memory overhead per JSON object (e.g. a key-value pair) requires an additional allocator pointer for type generalization [5]. Hence, the JSON map is not suited for big data storage, offering flexibility in exchange.
3. Keys are stored in an unordered map sorted in lexicographical order.

40.1 Migrating from Attribute

The ESMF_Info class is a replacement for the ESMF_Attribute class and is the preferred way of managing metadata attributes in ESMF moving forward. It is recommended that users migrate existing ESMF_Attribute calls to the new ESMF_Info API. The ESMF_Info class provides the backend for ESMF_Attribute since ESMF version 8.1. The ESMF_Attribute docs are located in appendix 57. In practice, users should experience no friction when migrating client code. Please email ESMF support in the case of a migration issue. Some structural changes to ESMF_Attribute did occur:

- Changed behavior when getting fixed-size lists. List size in storage must match the size of the outgoing list.
- Removed ability to use a default value with list gets.
- Removed `attPackInstanceName` from all interfaces.
- Removed `attcopyFlag` from all interfaces.
- Removed `ESMF_Attribute`-managed object linking.
- Modified `ESMF_AttributeAdd` to set the target key to a null JSON value.
- Modified `ESMF_AttributeSet` to not require an attribute added to an `ESMF_AttPack` be added through `ESMF_AttributeAdd` before setting.
- Removed support for attribute XML I/O.
- Removed ability to add multiple nested Attribute packages.
- Removed retrieval of "internal" ESMF object Attributes.

Below are examples for setting and getting an attribute using `ESMF_Info` and the legacy `ESMF_Attribute`. The `ESMF_Info` interfaces are not overloaded for ESMF object types but rather work off a handle retrieved via a get call.

40.1.1 Setting an Attribute

With `ESMF_Attribute`:

```
call ESMF_AttributeSet(array, "aKey", 15, rc=rc)
```

With `ESMF_Info`:

```
call ESMF_InfoGetFromHost(array, info, rc=rc)
call ESMF_InfoSet(info, "aKey", 15, rc=rc)
```

Notice that the legacy `ESMF_Attribute` API expects the usage of what was called an "Attribute Package". This essentially corresponds to a namespace similar to what `ESMF_Info` provides for keys via the JSON Pointer syntax (see 40.2). In the above `ESMF_AttributeSet()` call, without specification of `convention` and `purpose` arguments, the resulting JSON pointer of the key is "/ESMF/General/aKey". This is important to account for when mixing deprecated `ESMF_Attribute` calls with the `ESMF_Info` API.

40.1.2 Getting an Attribute

With `ESMF_Attribute`:

```
call ESMF_AttributeGet(array, "aKey", aKeyValue, rc=rc)
```

With `ESMF_Info`:

```
call ESMF_InfoGetFromHost(array, info, rc=rc)
call ESMF_InfoGet(info, "aKey", aKeyValue, rc=rc)
```

Notice again that the `ESMF_Attribute` API automatically prepends "/ESMF/General/" to the JSON pointer used for key in the absence of `convention` and `purpose` arguments.

40.2 Key Format Overview

A key in the `ESMF_Info` interface provides the location of a value to retrieve from the key-value storage. Keys in the `ESMF_Info` class use the JSON Pointer syntax [4]. A forward slash is prepended to string keys if it does not exist. Hence, `"aKey"` and `"/aKey"` are equivalent. Note the indexing aspect of the JSON Pointer syntax is not supported.

Some examples for valid "key" arguments:

- `altitude` :: A simple key argument with no nesting.
- `/altitude` :: A simple key argument with no nesting with the prepended pointer forward slash.
- `/altitude/height_above_mean_sea_level` :: A key for an attribute "height_above_mean_sea_level" nested in a map identified with key "altitude".

40.3 Usage and Examples

40.4 Class API

41 Time Manager Utility

The ESMF Time Manager utility includes software for time and date representation and calculations, model time advancement, and the identification of unique and periodic events. Since multi-component geophysical applications often require synchronization across the time management schemes of the individual components, the Time Manager's standard calendars and consistent time representation promote component interoperability.

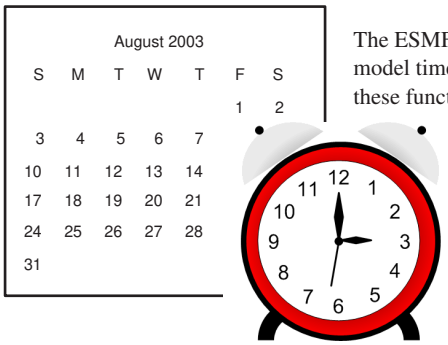
Key Features
Drift-free timekeeping through an integer-based internal time representation. Both integers and reals can be specified at the interface.
The ability to represent time as a rational fraction, to support exact timekeeping in applications that involve grid refinement.
Support for many calendar kinds, including user-customized calendars.
Support for both concurrent and sequential modes of component execution.
Support for varying and negative time steps.

41.1 Time Manager Classes

There are five ESMF classes that represent time concepts:

- **Calendar** A Calendar can be used to keep track of the date as an ESMF Gridded Component advances in time. Standard calendars (such as Gregorian and 360-day) and user-specified calendars are supported. Calendars can be queried for quantities such as seconds per day, days per month, and days per year.
- **Time** A Time represents a time instant in a particular calendar, such as November 28, 1964, at 7:31pm EST in the Gregorian calendar. The Time class can be used to represent the start and stop time of a time integration.
- **TimeInterval** TimeIntervals represent a period of time, such as 300 milliseconds. Time steps can be represented using TimeIntervals.

- **Clock** Clocks collect the parameters and methods used for model time advancement into a convenient package. A Clock can be queried for quantities such as start time, stop time, current time, and time step. Clock methods include incrementing the current time, and determining if it is time to stop.
- **Alarm** Alarms identify unique or periodic events by "ringing" - returning a true value - at specified times. For example, an Alarm might be set to ring on the day of the year when leaves start falling from the trees in a climate model.



The ESMF Time Manager utility includes software to manage model calendars, advance model time, and perform time and date calculations. The software classes that handle these functions are **Times**, **TimeIntervals**, **Clocks**, **Alarms**, and **Calendars**.

In the remainder of this section, we briefly summarize the functionality that the Time Manager classes provide. Detailed descriptions and usage examples precede the API listing for each class.

41.2 Calendar

An ESMF Calendar can be queried for seconds per day, days per month and days per year. The flexible definition of Calendars allows them to be defined for planetary bodies other than Earth. The set of supported calendars includes:

Gregorian The standard Gregorian calendar.

no-leap The Gregorian calendar with no leap years.

Julian The standard Julian date calendar.

Julian Day The standard Julian days calendar.

Modified Julian Day The Modified Julian days calendar.

360-day A 30-day-per-month, 12-month-per-year calendar.

no calendar Tracks only elapsed model time in hours, minutes, seconds.

See Section 42.1 for more details on supported standard calendars, and how to create a customized ESMF Calendar.

41.3 Time Instants and TimeIntervals

TimeIntervals and Time instants (simply called Times) are the computational building blocks of the Time Manager utility. TimeIntervals support operations such as add, subtract, compare size, reset value, copy value, and subdivide by a scalar. Times, which are moments in time associated with specific Calendars, can be incremented or decremented by TimeIntervals, compared to determine which of two Times is later, differenced to obtain the TimeInterval between two Times, copied, reset, and manipulated in other useful ways. Times support a host of different queries, both for values of individual Time components such as year, month, day, and second, and for derived values such as day of year, middle of current month and Julian day. It is also possible to retrieve the value of the hardware realtime clock in the form of a Time. See Sections 43.1 and 44.1, respectively, for use and examples of Times and TimeIntervals.

Since climate modeling, numerical weather prediction and other Earth and space applications have widely varying time scales and require different sorts of calendars, Times and TimeIntervals must support a wide range of time specifiers, spanning nanoseconds to years. The interfaces to these time classes are defined so that the user can specify a time using a combination of units selected from the list shown in Table 41.4.

41.4 Clocks and Alarms

Although it is possible to repeatedly step a Time forward by a TimeInterval using arithmetic on these basic types, it is useful to identify a higher-level concept to represent this function. We refer to this capability as a Clock, and include in its required features the ability to store the start and stop times of a model run, to check when time advancement should cease, and to query the value of quantities such as the current time and the time at the previous time step. The Time Manager includes a class with methods that return a true value when a periodic or unique event has taken place; we refer to these as Alarms. Applications may contain temporary or multiple Clocks and Alarms. Sections 45.1 and 46.1 describe the use of Clocks and Alarms in detail.

Table 3: Specifiers for Times and TimeIntervals

Unit	Meaning
<yy|yy_i8>	Year.
mm	Month of the year.
dd	Day of the month.
<d|d_i8|d_r8>	Julian or Modified Julian day.
<h|h_r8>	Hour.
<m|m_r8>	Minute.
<s|s_i8|s_r8>	Second.
<ms|ms_r8>	Millisecond.
<us|us_r8>	Microsecond.
<ns|ns_r8>	Nanosecond.
O	Time zone offset in integer number of hours and minutes.
<sN|sN_i8>	Numerator for times of the form s $+\frac{\text{sN}}{\text{sD}}$, where s is seconds and s, sN, and sD are integers. This format provides a mechanism for supporting exact behavior.
<sD|sD_i8	Denominator for times of the form s $+\frac{\text{sN}}{\text{sD}}$, where s is seconds and s, sN, and sD are integers.

41.5 Design and Implementation Notes

1. **Base TimeIntervals and Times on the same integer representation.** It is useful to allow both TimeIntervals and Times to inherit from a single class, BaseTime. In C++, this can be implemented by using inheritance. In Fortran, it can be implemented by having the derived types TimeIntervals and Times contain a derived type BaseTime. In both cases, the BaseTime class can be made private and invisible to the user.

   The result of this strategy is that Time Intervals and Times gain a consistent core representation of time as well a set of basic methods.

   The BaseTime class can be designed with a minimum number of elements to represent any required time. The design is based on the idea used in the real-time POSIX 1003.1b-1993 standard. That is, to represent time simply as a pair of integers: one for seconds (whole) and one for nanoseconds (fractional). These can then be converted at the interface level to any desired format.

   For ESMF, this idea can be modified and extended, in order to handle the requirements for a large time range (> 200,000 years) and to exactly represent any rational fraction, not just nanoseconds. To handle the large time range, a 64-bit or greater integer is used for whole seconds. Any rational fractional second is expressed using two additional integers: a numerator and a denominator. Both the whole seconds and fractional numerator are signed to handle negative time intervals and instants. For arithmetic consistency both must carry the same sign (both positive or both negative), except, of course, for zero values. The fractional seconds element (numerator) is bounded with respect to whole seconds. If the absolute value of the numerator becomes greater than or equal to the denominator, whole seconds are incremented or decremented accordingly and the numerator is reset to the remainder. Conversions are performed upon demand by interface methods within the TimeInterval and Time classes. This is done because different applications require different representations of time intervals and time instances. Floating point values as well as integers can be specified for the various time units in the interfaces, see Table 41.4. Floating point values are represented internally as integer-based rational fractions.

   The BaseTime class defines increment and decrement methods for basic TimeInterval calculations between Time instants. It is done here rather than in the Calendar class because it can be done with simple second-based

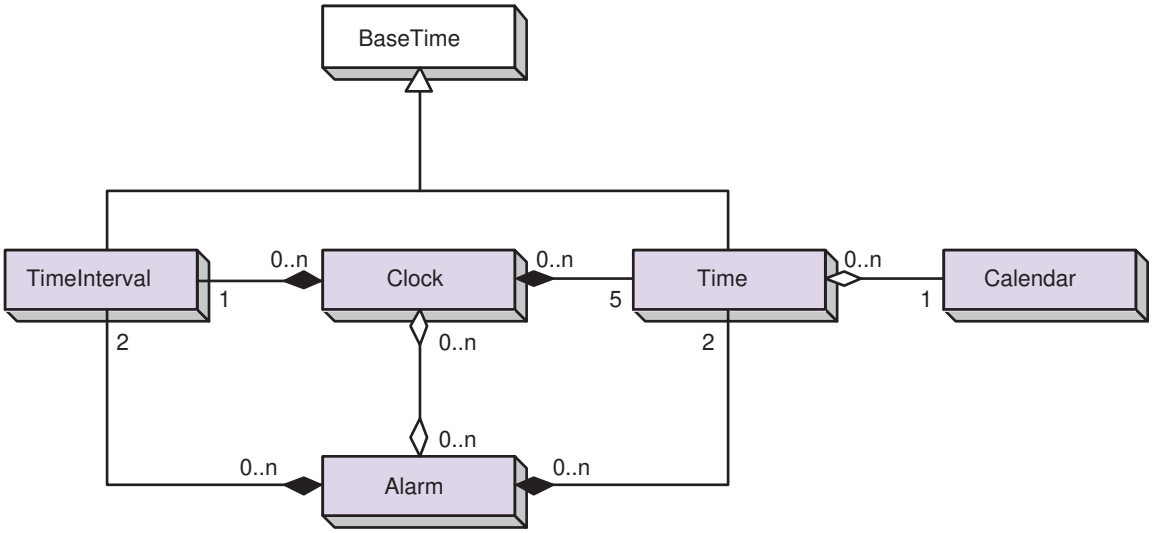
arithmetic that is calendar independent.

Comparison methods can also be defined in the BaseTime class. These perform equality/inequality, less than, and greater than comparisons between any two TimeIntervals or Times. These methods capture the common comparison logic between TimeIntervals and Times and hence are defined here for sharing.

2. **The Time class depends on a calendar.** The Time class contains an internal Calendar class. Upon demand by a user, the results of an increment or decrement operation are converted to user units, which may be calendar-dependent, via methods obtained from their internal Calendar.

41.6 Object Model

The following is a simplified UML diagram showing the structure of the Time Manager utility. See Appendix A, *A Brief Introduction to UML*, for a translation table that lists the symbols in the diagram and their meaning.



42 Calendar Class

42.1 Description

The Calendar class represents the standard calendars used in geophysical modeling: Gregorian, Julian, Julian Day, Modified Julian Day, no-leap, 360-day, and no-calendar. It also supports a user-customized calendar. Brief descriptions are provided for each calendar below. For more information on standard calendars, see [20] and [17].

42.2 Constants

42.2.1 ESMF_CALKIND

DESCRIPTION:
Supported calendar kinds.

The type of this flag is:

`type(ESMF_CalKind_Flag)`

The valid values are:

ESMF_CALKIND_360DAY *Valid range: machine limits*
In the 360-day calendar, there are 12 months, each of which has 30 days. Like the no-leap calendar, this is a simple approximation to the Gregorian calendar sometimes used by modelers.

ESMF_CALKIND_CUSTOM *Valid range: machine limits*
The user can set calendar parameters in the generic calendar.

ESMF_CALKIND_GREGORIAN *Valid range: 3/1/4801 BC to 10/29/292,277,019,914*
The Gregorian calendar is the calendar currently in use throughout Western countries. Named after Pope Gregory XIII, it is a minor correction to the older Julian calendar. In the Gregorian calendar every fourth year is a leap year in which February has 29 and not 28 days; however, years divisible by 100 are not leap years unless they are also divisible by 400. As in the Julian calendar, days begin at midnight.

ESMF_CALKIND_JULIAN *Valid range: 3/1/4713 BC to 4/24/292,271,018,333*
The Julian calendar was introduced by Julius Caesar in 46 B.C., and reached its final form in 4 A.D. The Julian calendar differs from the Gregorian only in the determination of leap years, lacking the correction for years divisible by 100 and 400 in the Gregorian calendar. In the Julian calendar, any year is a leap year if divisible by 4. Days are considered to begin at midnight.

ESMF_CALKIND_JULIANDAY *Valid range: +/- $1x10^{14}$*
Julian days simply enumerate the days and fraction of a day which have elapsed since the start of the Julian era, defined as beginning at noon on Monday, 1st January of year 4713 B.C. in the Julian calendar. Julian days, unlike the dates in the Julian and Gregorian calendars, begin at noon.

ESMF_CALKIND_MODJULIANDAY *Valid range: +/- $1x10^{14}$*
The Modified Julian Day (MJD) was introduced by space scientists in the late 1950's. It is defined as an offset from the Julian Day (JD):

MJD = JD - 2400000.5

The half day is subtracted so that the day starts at midnight.

ESMF_CALKIND_NOCALENDAR *Valid range: machine limits*
The no-calendar option simply tracks the elapsed model time in seconds.

ESMF_CALKIND_NOLEAP *Valid range: machine limits*
The no-leap calendar is the Gregorian calendar with no leap years - February is always assumed to have 28 days. Modelers sometimes use this calendar as a simple, close approximation to the Gregorian calendar.

42.3 Use and Examples

In most multi-component Earth system applications, the timekeeping in each component must refer to the same standard calendar in order for the components to properly synchronize. It therefore makes sense to create as few ESMF Calendars as possible, preferably one per application. A typical strategy would be to create a single Calendar at the start of an application, and use that Calendar in all subsequent calls that accept a Calendar, such as `ESMF_TimeSet`.

The following example shows how to set up an ESMF Calendar.

42.4 Restrictions and Future Work

1. **Months per year set to 12.** Due to the requirement of only Earth modeling, the number of months per year is hard-coded at 12. However, for easy modification, this is implemented via a C preprocessor #define MONTHS_PER_YEAR in ESMCI_Calendar.h.

2. **Calendar date conversions.** Date conversions are currently defined between the Gregorian, Julian, Julian Day, and Modified Julian Day calendars. Further research and work would need to be done to determine conversion algorithms with and between the other calendars: No Leap, 360 Day, and Custom.

3. **ESMF_CALKIND_CUSTOM.** Currently, there is no provision for a custom calendar to define a leap year rule, so `ESMF_CalendarIsLeapYear()` will always return `.false.` in this case. However, the arguments `daysPerYear`, `daysPerYearDn`, and `daysPerYearDd` in `ESMF_CalendarCreate()` and `ESMF_CalendarSet()` can be used to set a fractional number of days per year, for example, 365.25 = 365 25/100. Also, if further timekeeping precision is required, fractional and/or floating point `secondsPerDay` and `secondsPerYear` could be added to the interfaces `ESMF_CalendarCreate()`, `ESMF_CalendarSet()`, and `ESMF_CalendarGet()` and implemented.

42.5 Class API

43 Time Class

43.1 Description

A Time represents a specific point in time. In order to accommodate the range of time scales in Earth system applications, Times in the ESMF can be specified in many different ways, from years to nanoseconds. The Time interface is designed so that you select one or more options from a list of time units in order to specify a Time. The options for specifying a Time are shown in Table 41.4.

There are Time methods defined for setting and getting a Time, incrementing and decrementing a Time by a TimeInterval, taking the difference between two Times, and comparing Times. Special quantities such as the middle of the month and the day of the year associated with a particular Time can be retrieved. There is a method for returning the Time value as a string in the ISO 8601 format YYYY-MM-DDThh:mm:ss [15].

A Time that is specified in hours, minutes, seconds, or subsecond intervals does not need to be associated with a standard calendar; a Time whose specification includes time units of a day and greater must be. The ESMF representation of a calendar, the Calendar class, is described in Section 42.1. The `ESMF_TimeSet` method is used to initialize a Time as well as associate it with a Calendar. If a Time method is invoked in which a Calendar is necessary and one has not been set, the ESMF method will return an error condition.

In the ESMF the TimeInterval class is used to represent time periods. This class is frequently used in combination with the Time class. The Clock class, for example, advances model time by incrementing a Time with a TimeInterval.

43.2 Use and Examples

Times are most frequently used to represent start, stop, and current model times. The following examples show how to create, initialize, and manipulate `Time`.

43.3 Restrictions and Future Work

1. **Limits on size and resolution of Time.** The limits on the size and resolution of the time representation are based on the 64-bit integer types used. For seconds, a signed 64-bit integer will have a range of +/- 2^{63}-1, or +/- 9,223,372,036,854,775,807. This corresponds to a maximum size of +/- (2^{63}-1)/(86400 * 365.25) or +/- 292,271,023,045 years.

   For fractional seconds, a signed 64-bit integer will handle a resolution of +/- 2^{31}-1, or +/- 9,223,372,036,854,775,807 parts of a second.

43.4 Class API

44 TimeInterval Class

44.1 Description

A TimeInterval represents a period between time instants. It can be either positive or negative. Like the Time interface, the TimeInterval interface is designed so that you can choose one or more options from a list of time units in order to specify a TimeInterval. See Section 41.3, Table 41.4 for the available options.

There are TimeInterval methods defined for setting and getting a TimeInterval, for incrementing and decrementing a TimeInterval by another TimeInterval, and for multiplying and dividing TimeIntervals by integers, reals, fractions and other TimeIntervals. Methods are also defined to take the absolute value and negative absolute value of a TimeInterval, and for comparing the length of two TimeIntervals.

The class used to represent time instants in ESMF is Time, and this class is frequently used in operations along with TimeIntervals. For example, the difference between two Times is a TimeInterval.

When a TimeInterval is used in calculations that involve an absolute reference time, such as incrementing a Time with a TimeInterval, calendar dependencies may be introduced. The length of the time period that the TimeInterval represents will depend on the reference Time and the standard calendar that is associated with it. The calendar dependency becomes apparent when, for example, adding a TimeInterval of 1 day to the Time of February 28, 1996, at 4:00pm EST. In a 360 day calendar, the resulting date would be February 29, 1996, at 4:00pm EST. In a no-leap calendar, the result would be March 1, 1996, at 4:00pm EST.

TimeIntervals are used by other parts of the ESMF timekeeping system, such as Clocks (Section 45.1) and Alarms (Section 46.1).

44.2 Use and Examples

A typical use for a TimeInterval in a geophysical model is representation of the time step by which the model is advanced. Some models change the size of their time step as the model run progresses; this could be done by incrementing or decrementing the original time step by another TimeInterval, or by dividing or multiplying the time step by an integer value. An example of advancing model time using a TimeInterval representation of a time step is shown in Section 45.1.

The following brief example shows how to create, initialize and manipulate `TimeInterval`.

44.3 Restrictions and Future Work

1. **Limits on time span.** The limits on the time span that can be represented are based on the 64-bit integer types used. For seconds, a signed 64-bit integer will have a range of +/- 2^{63}-1, or +/- 9,223,372,036,854,775,807. This corresponds to a range of +/- (2^{63}-1)/(86400 * 365.25) or +/- 292,271,023,045 years.

   For fractional seconds, a signed 64-bit integer will handle a resolution of +/- 2^{31}-1, or +/- 9,223,372,036,854,775,807 parts of a second.

44.4 Class API

45 Clock Class

45.1 Description

The Clock class advances model time and tracks its associated date on a specified Calendar. It stores start time, stop time, current time, previous time, and a time step. It can also store a reference time, typically the time instant at which a simulation originally began. For a restart run, the reference time can be different than the start time, when the application execution resumes.

A user can call the `ESMF_ClockSet` method and reset the time step as desired.

A Clock also stores a list of Alarms, which can be set to flag events that occur at a specified time instant or at a specified time interval. See Section 46.1 for details on how to use Alarms.

There are methods for setting and getting the Times and Alarms associated with a Clock. Methods are defined for advancing the Clock's current time, checking if the stop time has been reached, reversing direction, and synchronizing with a real clock.

45.2 Constants

45.2.1 ESMF_DIRECTION

DESCRIPTION:

Specifies the time-stepping direction of a clock. Use with "direction" argument to methods `ESMF_ClockSet()` and `ESMF_ClockGet()`. Cannot be used with method `ESMF_ClockCreate()`, since it only initializes a clock in the default forward mode; a clock must be advanced (timestepped) at least once before reversing direction via `ESMF_ClockSet()`. This also holds true for negative timestep clocks which are initialized (created) with stopTime < startTime, since "forward" means timestepping from startTime towards stopTime (see `ESMF_DIRECTION_FORWARD` below).

"Forward" and "reverse" directions are distinct from positive and negative timesteps. "Forward" means timestepping in the direction established at `ESMF_ClockCreate()`, from startTime towards stopTime, regardless of the timestep sign. "Reverse" means timestepping in the opposite direction, back towards the clock's startTime, regardless of the timestep sign.

Clocks and alarms run in reverse in such a way that the state of a clock and its alarms after each time step is precisely replicated as it was in forward time-stepping mode. All methods which query clock and alarm state will return the same result for a given timeStep, regardless of the direction of arrival.

The type of this flag is:

`type(ESMF_Direction_Flag)`

The valid values are:

ESMF_DIRECTION_FORWARD Upon calling `ESMF_ClockAdvance()`, the clock will timestep from its startTime toward its stopTime. This is the default direction. A user can use either `ESMF_ClockIsStopTime()` or `ESMF_ClockIsDone()` methods to determine when stopTime is reached. This forward behavior also holds for negative timestep clocks which are initialized (created) with stopTime < startTime.

ESMF_DIRECTION_REVERSE Upon calling `ESMF_ClockAdvance()`, the clock will timestep backwards toward its startTime. Use method `ESMF_ClockIsDone()` to determine when startTime is reached. This reverse

behavior also holds for negative timestep clocks which are initialized (created) with stopTime < startTime.

45.3 Use and Examples

The following is a typical sequence for using a Clock in a geophysical model.

At initialize:

- Set a Calendar.
- Set start time, stop time and time step as Times and Time Intervals.
- Create and Initialize a Clock using the start time, stop time and time step.
- Define Times and Time Intervals associated with special events, and use these to set Alarms.

At run:

- Advance the Clock, checking for ringing alarms as needed.
- Check if it is time to stop.

At finalize:

- Since Clocks and Alarms are deep classes, they need to be explicitly destroyed at finalization. Times and TimeIntervals are lightweight classes, so they don't need explicit destruction.

The following code example illustrates Clock usage.

45.4 Restrictions and Future Work

1. **Alarm list allocation factor** The alarm list within a clock is dynamically allocated automatically, 200 alarm references at a time. This constant is defined in both Fortran and C++ with a #define for ease of modification.
2. **Clock variable timesteps in reverse**

   In order for a clock with variable timesteps to be run in `ESMF_DIRECTION_REVERSE`, the user must supply those timesteps to `ESMF_ClockAdvance()`. Essentially, the user must save the timesteps while in forward mode. In a future release, the Time Manager will assume this responsibility by saving the clock state (including the timeStep) at every timestep while in forward mode.

45.5 Class API

46 Alarm Class

46.1 Description

The Alarm class identifies events that occur at specific Times or specific TimeIntervals by returning a true value at those times or subsequent times, and a false value otherwise.

46.2 Constants

46.2.1 ESMF_ALARMLIST

DESCRIPTION:
Specifies the characteristics of Alarms that populate a retrieved Alarm list.

The type of this flag is:

`type(ESMF_AlarmList_Flag)`

The valid values are:

ESMF_ALARMLIST_ALL All alarms.

ESMF_ALARMLIST_NEXTRINGING Alarms that will ring before or at the next timestep.

ESMF_ALARMLIST_PREVRINGING Alarms that rang at or since the last timestep.

ESMF_ALARMLIST_RINGING Only ringing alarms.

46.3 Use and Examples

Alarms are used in conjunction with Clocks (see Section 45.1). Multiple Alarms can be associated with a Clock. During the `ESMF_ClockAdvance()` method, a Clock iterates over its internal Alarms to determine if any are ringing. Alarms ring when a specified Alarm time is reached or exceeded, taking into account whether the time step is positive or negative. In `ESMF_DIRECTION_REVERSE` (see Section 45.1), alarms ring in reverse, i.e., they begin ringing when they originally ended, and end ringing when they originally began. On completion of the time advance call, the Clock optionally returns a list of ringing alarms.

Each ringing Alarm can then be processed using Alarm methods for identifying, turning off, disabling or resetting the Alarm.

Alarm methods are defined for obtaining the ringing state, turning the ringer on/off, enabling/disabling the Alarm, and getting/setting associated times.

The following example shows how to set and process Alarms.

46.4 Restrictions and Future Work

1. **Alarm list allocation factor** The alarm list within a clock is dynamically allocated automatically, 200 alarm references at a time. This constant is defined in both Fortran and C++ with a #define for ease of modification.

2. **Sticky alarm end times in reverse** For sticky alarms, there is an implicit limitation that in order to properly reverse timestep through a ring end time, that time must have already been traversed in the forward direction. This is due to the fact that the Time Manager cannot predict when user code will call `ESMF_AlarmRingerOff()`. An error message will be logged when this limitation is not satisfied.

3. **Sticky alarm ring interval in reverse**

   For repeating sticky alarms, it is currently assumed that the ringInterval is constant, so that only the time of the last call to `ESMF_AlarmRingerOff()` is saved. In `ESMF_DIRECTION_REVERSE`, this information is used to turn sticky alarms back on. In a future release, ringIntervals will be allowed to be variable, by saving alarm state at every timestep.

46.5 Design and Implementation Notes

The Alarm class is designed as a deep, dynamically allocatable class, based on a pointer type. This allows for both indirect and direct manipulation of alarms. Indirect alarm manipulation is where ESMF_Alarm API methods, such as ESMF_AlarmRingerOff(), are invoked on alarm references (pointers) returned from ESMF_Clock queries such as "return ringing alarms." Since the method is performed on an alarm reference, the actual alarm held by the clock is affected, not just a user's local copy. Direct alarm manipulation is the more common case where alarm API methods are invoked on the original alarm objects created by the user.

For consistency, the ESMF_Clock class is also designed as a deep, dynamically allocatable class.

An additional benefit from this approach is that Clocks and Alarms can be created and used from anywhere in a user's code without regard to the scope in which they were created. In contrast, statically created Alarms and Clocks would disappear if created within a user's routine that returns, whereas dynamically allocated Alarms and Clocks will persist until explicitly destroyed by the user.

46.6 Class API

47 Config Class

47.1 Description

ESMF Configuration Management is based on NASA DAO's Inpak 90 package, a Fortran 90 collection of routines/functions for accessing *Resource Files* in ASCII format.The package is optimized for minimizing formatted I/O, performing all of its string operations in memory using Fortran intrinsic functions.

47.1.1 Package history

The ESMF Configuration Management Package was evolved by Leonid Zaslavsky and Arlindo da Silva from Ipack90 package created by Arlindo da Silva at NASA DAO.

Back in the 70's Eli Isaacson wrote IOPACK in Fortran 66. In June of 1987 Arlindo da Silva wrote Inpak77 using Fortran 77 string functions; Inpak 77 is a vastly simplified IOPACK, but has its own goodies not found in IOPACK. Inpak 90 removes some obsolete functionality in Inpak77, and parses the whole resource file in memory for performance.

47.1.2 Resource files

A *Resource File (RF)* is a text file consisting of list of *label-value* pairs. There is a buffer limit of 256,000 characters for the entire Resource File. Each *label* is limited to 1,000 characters. Each label should be followed by some data, the *value*. An example Resource File follows. It is the file used in the example below.

```
# This is an example Resource File.
# It contains a list of <label,value> pairs.
# The colon after the label is required.

# The values after the label can be an list.
# Multiple types are authorized.

 my_file_names:         jan87.dat jan88.dat jan89.dat  # all strings
 constants:             3.1415   25                    # float and integer
 my_favorite_colors:    green blue 022


# Or, the data can be a list of single value pairs.
# It is simplier to retrieve data in this format:

 radius_of_the_earth:   6.37E6
 parameter_1:           89
 parameter_2:           78.2
 input_file_name:       dummy_input.nc


# Or, the data can be located in a table using the following
# syntax:

 my_table_name::
  1000     3000     263.0
   925     3000     263.0
   850     3000     263.0
   700     3000     269.0
   500     3000     287.0
   400     3000     295.8
   300     3000     295.8
 ::
```

Note that the colon after the label is required and that the double colon is required to declare tabular data.

Resource files are intended for random access (except between ::'s in a table definition). This means that order in which a particular *label-value* pair is retrieved is not dependent upon the original order of the pairs. The only exception to this, however, is when the same *label* appears multiple times within the Resource File.

47.2 Use and Examples

47.3 Class API

48 HConfig Class

48.1 Description

The ESMF HConfig class implements a hierarchical configuration facility that is compatible with YAML Ain't Markup Language (YAML™). ESMF HConfig can be understood as a Fortran interface to YAML. However, no claim is made that *all* YAML language features are supported in their entirety.

The purpose of the HConfig class under ESMF is to provide a migration path toward more standard configuration management for ESMF applications. To this end ESMF_HConfig integrates with the traditional ESMF_Config class. Through this integration the traditional Config class API offers basic access to YAML configuration files, in addition to providing backward compatible support of the traditional config file format. This is discussed in more detail in the Config class section. For more complete YAML support, applications are encouraged to migrate to the HConfig API discussed in this section.

48.2 Constants

48.2.1 ESMF_HCONFIGMATCH

DESCRIPTION:
Indicates the level to which two HConfig variables match.

The type of this flag is:

`type(ESMF_HConfigMatch_Flag)`

The valid values in ascending order are:

ESMF_HCONFIGMATCH_INVALID: Indicates a non-valid matching level. One or both HConfig objects are invalid.

ESMF_HCONFIGMATCH_NONE: The lowest valid level of HConfig matching. This indicates that the HConfig objects are valid, but their YAML representation does not match.

ESMF_HCONFIGMATCH_EXACT: There is an exact match between the YAML representation of both HConfig objects. They may or may not be aliases to the same object in memory.

ESMF_HCONFIGMATCH_ALIAS: Both HConfig variables are aliases to the exact same HConfig object in memory.

48.3 Use and Examples

The following examples demonstrate how a user typically interacts with the HConfig API. The HConfig class introduces two derived types:

- `ESMF_HConfig`

- ESMF_HConfigIter

ESMF_HConfig objects can be created explicitly by the user, or they can be accessed from an existing ESMF_Config object, e.g. queried from a Component. They can play a number of roles when interacting with a HConfig hierarchy:

1. The root node of the entire hierarchy. In YAML terminology, this refers to a *document*.
2. *Any* node within the hierarchy.
3. Collection of hierarchies, i.e. a set of YAML *documents*.

ESMF_HConfigIter objects are iterators, *referencing* a specific node within the hierarchy. They are created from ESMF_HConfig objects. The iterator approach allows convenient sequential traversal of a particular location in the HConfig hierarchy. There are *two* flavors of iterators in HConfig: *sequence* and *map* iterators. Both are represented by the same ESMF_HConfigIter derived type, and the distinction is made at run-time.

Notice that there are redundancies built into the HConfig API, where different ways are available to achieve the same goal. This is mostly done for convenience, allowing the user to pick the approach most suitable to their needs.

For instance, while it can be convenient to use iterators in some cases, in others, it might be more appropriate to access elements directly by *index* (for sequences) or *key* (for maps). Both options are available.

48.4 Restrictions and Future Work

- The YAML Core schema, which is an extension of the JSON schema, is implemented and used to resolve non-specific tags under HConfig. There is currently no mechanism implemented to switch to a different schema for tag resolution.
- Currently the only available removal method for HConfig *map* objects requires that *keys* be simple scalar strings.
- There is currently no method implemented that allows setting of tags from from the API.

48.5 Design and Implementation Notes

The ESMF HConfig class is implemented on top of YAML-CPP (https://github.com/jbeder/yaml-cpp). A copy of YAML-CPP is included in the ESMF source tree under ./src/prologue/yaml-cpp. It is used by a number of ESMF/NUOPC functions, including HConfig.

48.6 Class API

49 Log Class

49.1 Description

The Log class consists of a variety of methods for writing error, warning, and informational messages to files. A default Log is created at ESMF initialization. Other Logs can be created later in the code by the user. Most Log methods take a Log as an optional argument and apply to the default Log when another Log is not specified. A set of standard return codes and associated messages are provided for error handling.

Log provides capabilities to store message entries in a buffer, which is flushed to a file, either when the buffer is full, or when the user calls an ESMF_LogFlush() method. Currently, the default is for the Log to flush after every ten entries. This can easily be changed by using the ESMF_LogSet() method and setting the maxElements property to another value. The ESMF_LogFlush() method is automatically called when the program exits by any means (program completion, halt on error, or when the Log is closed).

The user has the capability to abort the program on conditions such as an error or on a warning by using the ESMF_LogSet() method with the logmsgAbort argument. For example if the logmsgAbort array is set to (ESMF_LOGMSG_ERROR,ESMF_LOGMSG_WARNING), the program will stop on any and all warning or errors. When the logmsgAbort argument is set to ESMF_LOGMSG_ERROR, the program will only abort on errors. Lastly, the user can choose to never abort by using ESMF_LOGMSG_NONE; this is the default.

Log will automatically put the PET number into the Log. Also, the user can either specify ESMF_LOGKIND_SINGLE which writes all the entries to a single Log or ESMF_LOGKIND_MULTI which writes entries to multiple Logs according to the PET number. To distinguish Logs from each other when using ESMF_LOGKIND_MULTI, the PET number (in the format PETx.) will be prepended to the file name where x is the PET number.

Opening multiple log files and writing log messages from all the processors may affect the application performance while running on a large number of processors. For that reason, ESMF_LOGKIND_NONE is provided to switch off the Log capability. All the Log methods have no effect in the ESMF_LOGKIND_NONE mode.

A tracing capability may be enabled by setting the trace flag by using the ESMF_LogSet() method. When tracing is enabled, calls to methods such as ESMF_LogFoundError, ESMF_LogFoundAllocError, and ESMF_LogFoundDeallocError are logged in the default log file. This can result in voluminous output. It is typically used only around areas of code which are being debugged.

Other options that are planned for Log are to adjust the verbosity of output, and to optionally write to stdout instead of file(s).

49.2 Constants

49.2.1 ESMF_LOGERR

The valid values are:

ESMF_LOGERR_PASSTHRU A named character constant, with a predefined generic error message, that can be used for the msg argument in any ESMF_Log routine. The message indicated by this named constant is "*Passing error in return code*."

49.2.2 ESMF_LOGKIND

DESCRIPTION:
Specifies a single log file, multiple log files (one per PET), or no log files.

The type of this flag is:

type(ESMF_LogKind_Flag)

The valid values are:

ESMF_LOGKIND_SINGLE Use a single log file, combining messages from all of the PETs. Not supported on some platforms.

ESMF_LOGKIND_MULTI Use multiple log files — one per PET.

ESMF_LOGKIND_MULTI_ON_ERROR Use multiple log files — one per PET. A log file is only opened when a message of type `ESMF_LOGMSG_ERROR` is encountered.

ESMF_LOGKIND_NONE Do not issue messages to a log file.

49.2.3 ESMF_LOGMSG

DESCRIPTION:

Specifies a message level

The type of this flag is:

`type(ESMF_LogMsg_Flag)`

The valid values are:

ESMF_LOGMSG_INFO Informational messages

ESMF_LOGMSG_WARNING Warning messages

ESMF_LOGMSG_ERROR Error messages

ESMF_LOGMSG_TRACE Trace messages

ESMF_LOGMSG_DEBUG DEBUG messages

ESMF_LOGMSG_JSON JSON format messages

Valid predefined named array constant values are:

ESMF_LOGMSG_ALL All messages

ESMF_LOGMSG_NONE No messages

ESMF_LOGMSG_NOTRACE All messages EXCEPT trace messages

49.3 Use and Examples

By default `ESMF_Initialize()` opens a default Log in `ESMF_LOGKIND_MULTI` mode. ESMF handles the initialization and finalization of the default Log so the user can immediately start using it. If additional Log objects are desired, they must be explicitly created or opened using `ESMF_LogOpen()`.

`ESMF_LogOpen()` requires a Log object and filename argument. Additionally, the user can specify single or multi Logs by setting the `logkindflag` property to `ESMF_LOGKIND_SINGLE` or `ESMF_LOGKIND_MULTI`. This is useful as the PET numbers are automatically added to the Log entries. A single Log will put all entries, regardless of PET number, into a single log while a multi Log will create multiple Logs with the PET number prepended to the filename and all entries will be written to their corresponding Log by their PET number.

By default, the Log file is not truncated at the start of a new run; it just gets appended each time. Future functionality may include an option to either truncate or append to the Log file.

In all cases where a Log is opened, a Fortran unit number is assigned to a specific Log. A Log is assigned an unused unit number using the algorithm described in the ESMF_IOUnitGet() method.

The user can then set or get options on how the Log should be used with the ESMF_LogSet() and ESMF_LogGet() methods. These are partially implemented at this time.

Depending on how the options are set, ESMF_LogWrite() either writes user messages directly to a Log file or writes to a buffer that can be flushed when full or by using the ESMF_LogFlush() method. The default is to flush after every ten entries because maxElements is initialized to ten (which means the buffer reaches its full state after every ten writes and then flushes).

A message filtering option may be set with ESMF_LogSet() so that only selected message types are actually written to the log. One key use of this feature is to allow placing informational log write requests into the code for debugging or tracing. Then, when the informational entries are not needed, the messages at that level may be turned off — leaving only warning and error messages in the logs.

For every ESMF_LogWrite(), a time and date stamp is prepended to the Log entry. The time is given in microsecond precision. The user can call other methods to write to the Log. In every case, all methods eventually make a call implicitly to ESMF_LogWrite() even though the user may never explicitly call it.

When calling ESMF_LogWrite(), the user can supply an optional line, file and method. These arguments can be passed in explicitly or with the help of cpp macros. In the latter case, a define for an ESMF_FILENAME must be placed at the beginning of a file and a define for ESMF_METHOD must be placed at the beginning of each method. The user can then use the ESMF_CONTEXT cpp macro in place of line, file and method to insert the parameters into the method. The user does not have to specify line number as it is a value supplied by cpp.

An example of Log output is given below running with logkindflag property set to ESMF_LOGKIND_MULTI (default) using the default Log:

(Log file PET0.ESMF_LogFile)

```
20041105 163418.472210 INFO      PET0     Running with ESMF Version 2.2.1
```

(Log file PET1.ESMF_LogFile)

```
20041105 163419.186153 ERROR     PET1     ESMF_Field.F90              812
ESMF_FieldGet No Grid or Bad Grid attached to Field
```

The first entry shows date and time stamp. The time is given in microsecond precision. The next item shown is the type of message (INFO in this case). Next, the PET number is added. Lastly, the content is written.

The second entry shows something slightly different. In this case, we have an ERROR. The method name (ESMF_Field.F90) is automatically provided from the cpp macros as well as the line number (812). Then the content of the message is written.

When done writing messages, the default Log is closed by calling ESMF_LogFinalize() or ESMF_LogClose() for user created Logs. Both methods will release the assigned unit number.

49.4 Restrictions and Future Work

1. **Line, file and method are only available when using the C preprocessor** Message writing methods are expanded using the ESMF macro ESMF_CONTEXT that adds the predefined symbolic constants __LINE__ and __FILE__ (or the ESMF constant ESMF_FILENAME if defined) and the ESMF constant ESMF_METHOD to the argument list. Using these constants, we can associate a file name, line number and method name with

the message. If the CPP preprocessor is not used, this expansion will not be done and hence the ESMF macro ESMF_CONTEXT can not be used, leaving the file name, line number and method out of the Log text.

2. **Get and set methods are partially implemented.** Currently, the `ESMF_LogGet()` and `ESMF_LogSet()` methods are partially implemented.

3. **Log only appends entries.** All writing to the Log is appended rather than overwriting the Log. Future enhancements include the option to either append to an existing Log or overwrite the existing Log.

4. **Avoiding conflicts with the default Log.**

   The private methods `ESMF_LogInitialize()` and `ESMF_LogFinalize()` are called during `ESMF_Initialize()` and `ESMF_Finalize()` respectively, so they do not need to be called if the default Log is used. If a new Log is required, `ESMF_LogOpen()` is used with a new Log object passed in so that there are no conflicts with the default Log.

5. **ESMF_LOGKIND_SINGLE does not work properly.** When the `ESMF_LogKind_Flag` is set to `ESMF_LOGKIND_SINGLE`, different system may behave differently. The log messages from some processors may be lost or overwritten by other processors. Users are advised not to use this mode. The MPI-based I/O will be implemented to fix the problem in the future release.

49.5 Design and Implementation Notes

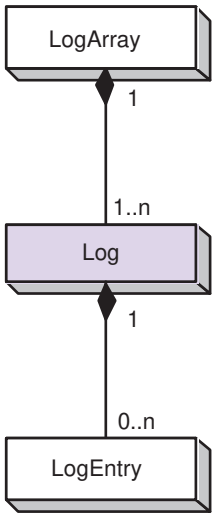
1. The Log class was implemented in Fortran and uses the Fortran I/O libraries when the class methods are called from Fortran. The C/C++ Log methods use the Fortran I/O library by calling utility functions that are written in Fortran. These utility functions call the standard Fortran write, open and close functions. At initialization an `ESMF_LOG` is created. The `ESMF_LOG` stores information for a specific Log file. When working with more than one Log file, multiple `ESMF_LOG`'s are required (one `ESMF_LOG` for each Log file). For each Log, a handle is returned through the `ESMF_LogInitialize` method for the default log or `ESMF_LogOpen` for a user created log. The user can specify single or multi logs by setting the `logkindflag` property in the `ESMF_LogInitialize` or `ESMF_Open` method to `ESMF_LOGKIND_SINGLE` or `ESMF_LOGKIND_MULTI`. Similarly, the user can set the `logkindflag` property for the default Log with the `ESMF_Initialize` method call. The `logkindflag` is useful as the PET numbers are automatically added to the log entries. A single log will put all entries, regardless of PET number, into a single log while a multi log will create multiple logs with the PET number prepended to the filename and all entries will be written to their corresponding log by their PET number.

   The properties for a Log are set with the `ESMF_LogSet()` method and retrieved with the `ESMF_LogGet()` method.

   Additionally, buffering is enabled. Buffering allows `ESMF` to manage output data streams in a desired way. Writing to the buffer is transparent to the user because all the Log entries are handled automatically by the `ESMF_LogWrite()` method. All the user has to do is specify the buffer size (the default is ten) by setting the `maxElements` property. Every time the `ESMF_LogWrite()` method is called, a LogEntry element is populated with the `ESMF_LogWrite()` information. When the buffer is full (i.e., when all the LogEntry elements are populated), the buffer will be flushed and all the contents will be written to file. If buffering is not needed, that is `maxElements=1` or `flushImmediately=ESMF_TRUE`, the `ESMF_LogWrite()` method will immediately write to the Log file(s).

49.6 Object Model

The following is a simplified UML diagram showing the structure of the Log class. See Appendix A, *A Brief Introduction to UML*, for a translation table that lists the symbols in the diagram and their meaning.



49.7 Class API

50 DELayout Class

50.1 Description

The DELayout class provides an additional layer of abstraction on top of the Virtual Machine (VM) layer. DELayout does this by introducing DEs (Decomposition Elements) as logical resource units. The DELayout object keeps track of the relationship between its DEs and the resources of the associated VM object.

The relationship between DEs and VM resources (PETs (Persistent Execution Threads) and VASs (Virtual Address Spaces)) contained in a DELayout object is defined during its creation and cannot be changed thereafter. There are, however, a number of hint and specification arguments that can be used to shape the DELayout during its creation.

Contrary to the number of PETs and VASs contained in a VM object, which are fixed by the available resources, the number of DEs contained in a DELayout can be chosen freely to best match the computational problem or other design criteria. Creating a DELayout with less DEs than there are PETs in the associated VM object can be used to share resources between decomposed objects within an ESMF component. Creating a DELayout with more DEs than there are PETs in the associated VM object can be used to evenly partition the computation over the available resources.

The simplest case, however, is where the DELayout contains the same number of DEs as there are PETs in the associated VM context. In this case the DELayout may be used to re-label the hardware and operating system resources held by the VM. For instance, it is possible to order the resources so that specific DEs have best available communication paths. The DELayout will map the DEs to the PETs of the VM according to the resource details provided by the VM instance.

Furthermore, general DE to PET mapping can be used to offer computational resources with finer granularity than the VM does. The DELayout can be queried for computational and communication capacities of DEs and DE pairs, respectively. This information can be used to best utilize the DE resources when partitioning the computational problem. In combination with other ESMF classes, general DE to PET mapping can be used to realize cache blocking, communication hiding and dynamic load balancing.

Finally, the DELayout layer offers primitives that allow a work queue style dynamic load balancing between DEs.

50.2 Constants

50.2.1 ESMF_PIN

DESCRIPTION:
Specifies which VM resource DEs are pinned to (PETs, VASs, SSIs).

The type of this flag is:

`type(ESMF_Pin_Flag)`

The valid values are:

ESMF_PIN_DE_TO_PET Pin DEs to PETs. Only the owning PET has access to a DE.

ESMF_PIN_DE_TO_VAS Pin DEs to virtual address spaces (VAS). DEs are accessible from all PETs within the same VAS.

ESMF_PIN_DE_TO_SSI Pin DEs to single system images (SSI) - typically shared memory nodes. DEs are accessible from all PETs within the same SSI. The memory allocation between different DEs is allowed to be non-contiguous.

ESMF_PIN_DE_TO_SSI_CONTIG Same as `ESMF_PIN_DE_TO_SSI`, but the shared memory allocation across DEs located on the same SSI must be contigous throughout.

50.2.2 ESMF_SERVICEREPLY

DESCRIPTION:
Reply when a PET offers to service a DE.

The type of this flag is:

`type(ESMF_ServiceReply_Flag)`

The valid values are:

ESMF_SERVICEREPLY_ACCEPT The service offer has been accepted. The PET is expected to service the DE.

ESMF_SERVICEREPLY_DENY The service offer has been denied. The PET is expected to not service the DE.

50.3 Use and Examples

The following examples demonstrate how to create, use and destroy DELayout objects.

50.4 Restrictions and Future Work

50.5 Design and Implementation Notes

The DELayout class is a light weight object. It stores the DE to PET and VAS mapping for all DEs within all PET instances and a list of local DEs for each PET instance. The DELayout does not store the computational and communication weights optionally provided as arguments to the create method. These hints are only used during create while they are available in user owned arrays.

50.6 Class API

51 VM Class

51.1 Description

The ESMF VM (Virtual Machine) class is a generic representation of hardware and system software resources. There is exactly one VM object per ESMF Component, providing the execution environment for the Component code. The VM class handles all resource management tasks for the Component class and provides a description of the underlying configuration of the compute resources used by a Component.

In addition to resource description and management, the VM class offers the lowest level of ESMF communication methods. The VM communication calls are very similar to MPI. Data references in VM communication calls must be provided as raw, language-specific, one-dimensional, contiguous data arrays. The similarity between VM and MPI communication calls is striking and there are many equivalent point-to-point and collective communication calls. However, unlike MPI, the VM communication calls support communication between threaded PETs in a completely transparent fashion.

Many ESMF applications do not interact with the VM class directly very much. The resource management aspect is wrapped completely transparent into the ESMF Component concept. Often the only reason that user code queries a Component object for the associated VM object is to inquire about resource information, such as the `localPet` or the `petCount`. Further, for most applications the use of higher level communication APIs, such as provided by Array and Field, are much more convenient than using the low level VM communication calls.

The basic elements of a VM are called PETs, which stands for Persistent Execution Threads. These are equivalent to OS threads with a lifetime of at least that of the associated component. All VM functionality is expressed in terms of PETs. In the simplest, and most common case, a PET is equivalent to an MPI process. However, ESMF also supports multi-threading, where multiple PETs run as Pthreads inside the same virtual address space (VAS).

The resource management functions of the VM class become visible when a component, or the driver code, creates sub-components. Section 16.4.3 discusses this aspect from the Superstructure perspective and provides links to the relevant Component examples in the documentation.

There are two parts to resource management, the parent and the child. When the parent component creates a child component, the parent VM object provides the resources on which the child is created with `ESMF_GridCompCreate()` or `ESMF_CplCompCreate()`. The optional `petList` argument to these calls limits the resources that the parent gives to a specific child. The child component, may specify - during its optional `ESMF_<Grid/Cpl>CompSetVM()` method - how it wants to arrange the inherited resources in its own VM. After this, all standard ESMF methods of the Component, including `ESMF_<Grid/Cpl>CompSetServices()`, will execute in the child VM. Notice that the `ESMF_<Grid/Cpl>CompSetVM()` routine, although part of the child Component, must execute *before* the child VM has been started up. It runs in the parent VM context. The

child VM is created and started up just before the user-written set services routine, specified as an argument to ESMF_<Grid/Cpl>CompSetServices(), is entered.

51.2 Constants

51.2.1 ESMF_VMEPOCH

DESCRIPTION:
Specifies the kind of VM Epoch being entered.

The type of this flag is:

type(ESMF_VMEpoch_Flag)

The valid values are:

ESMF_VMEPOCH_NONE An epoch wihout special behavior.

ESMF_VMEPOCH_BUFFER This option must only be used for parts of the code with distinct sending and receiving PETs, i.e. where no PETs are both sender and receiver. All non-blocking messages are being buffered. A single message is sent between unique pairs of src-dst PETs. This can significantly improve performance for cases with a large imbalance in the number of sending versus receiving PETs. The extra buffering also improves the overall asynchronous behavior between the sending and receiving side.

51.3 Use and Examples

The concept of the ESMF Virtual Machine (VM) is so fundamental to the framework that every ESMF application uses it. However, for many user applications the VM class is transparently hidden behind the ESMF Component concept and higher data classes (e.g. Array, Field). The interaction between user code and VM is often only indirect. The following examples provide an overview of where the VM class can come into play in user code.

51.4 Restrictions and Future Work

1. **Only array section syntax that leads to contiguous sub sections is supported**. The source and destination arguments in VM communication calls must reference contiguous data arrays. Fortran array sections are not guaranteed to be contiguous in all cases.

2. **Non-blocking** Reduce() **operations *not* implemented.** None of the reduce communication calls have an implementation for the non-blocking feature. This affects:

   - ESMF_VMAllFullReduce(),
   - ESMF_VMAllReduce(),
   - ESMF_VMReduce().

3. **Limitations when using** mpiuni **mode.** In mpiuni mode non-blocking communications are limited to one outstanding message per source-destination PET pair. Furthermore, in mpiuni mode the message length must be smaller than the internal ESMF buffer size.

4. **Alternative communication paths not accessible.** All user accessible VM communication calls are currently implemented using MPI-1.2. VM's implementation of alternative communication techniques, such as shared

memory between threaded PETs and POSIX IPC between PETs located on the same single system image, are currently inaccessible to the user. (One exception to this is the `mpiuni` case for which the VM automatically utilizes a shared memory path.)

5. **Data arrays in VM comm calls are *assumed shape* with rank=1.** Currently all dummy arrays in VM comm calls are defined as *assumed shape* arrays of rank=1. The motivation for this choice is that the use of assumed shape dummy arrays guards against the Fortran copy in/out problem. However it may not be as flexible as desired from the user perspective. Alternatively all dummy arrays could be defined as *assumed size* arrays, as it is done in most MPI implementations, allowing arrays of various rank to be passed into the comm methods. Arrays of higher rank can be passed into the current interfaces using Fortran array syntax. This approach is explained in section **??**.

6. **Limitations when using VMEpoch.** Using a blocking collective call (e.g. `ESMF_VMBroadcast()`, the `MPI_Bcast()` used by `ESMF_InfoBroadcast()`, etc.) within the region enclosed by `ESMF_VMEpochEnter()` and `ESMF_VMEpochExit()` will result in a deadlock.

51.5 Design and Implementation Notes

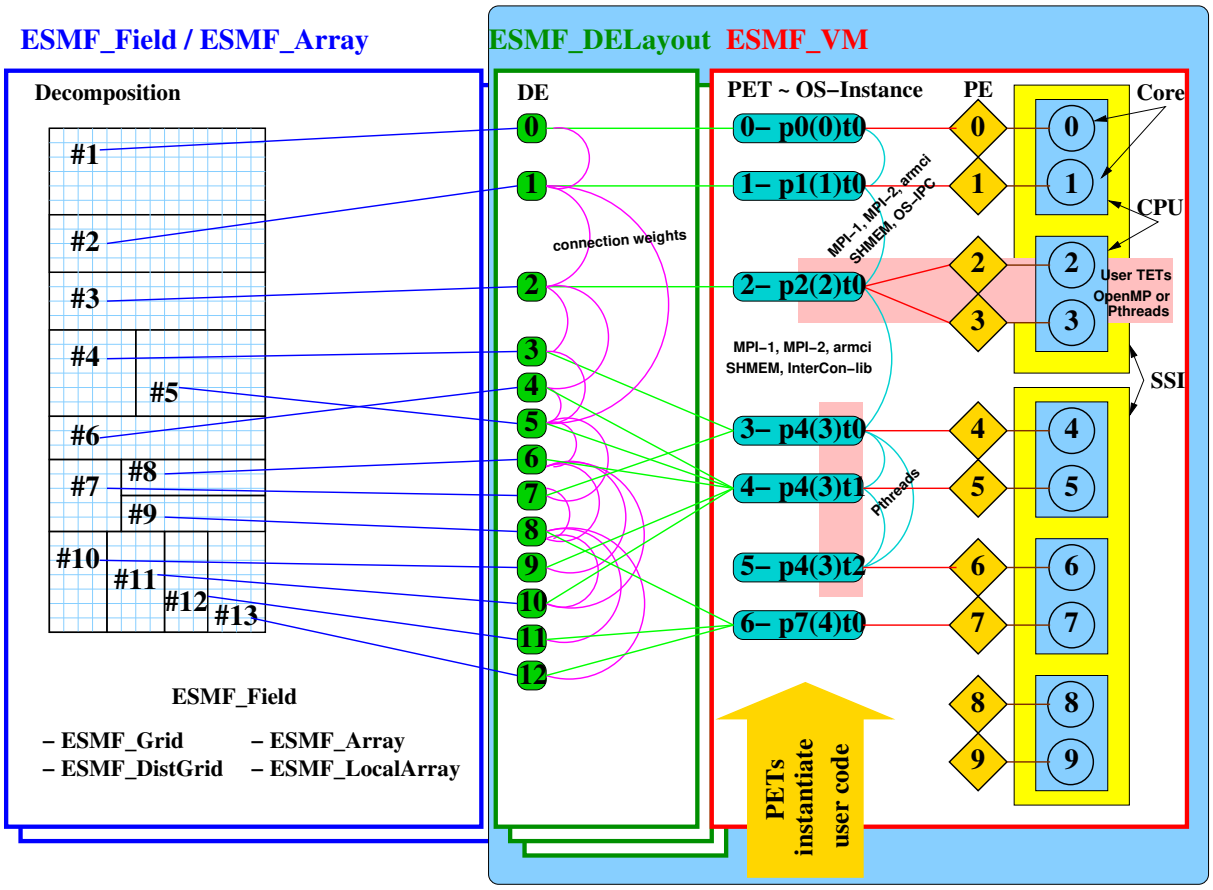
The VM class provides an additional layer of abstraction on top of the POSIX machine model, making it suitable for HPC applications. There are four key aspects the VM class deals with.

1. Encapsulation of hardware and operating system details within the concept of Persistent Execution Threads (PETs).

2. Resource management in terms of PETs with a guard against over-subscription.

3. Topological description of the underlying configuration of the compute resources in terms of PETs.

4. Transparent communication API for point-to-point and collective PET-based primitives, hiding the many different communication channels and offering best possible performance.



Definition of terms used in the diagram

- PE: A processing element (PE) is an alias for the smallest physical processing unit available on a particular hardware platform. In the language of today's microprocessor architecture technology a PE is identical to a core, however, if future microprocessor designs change the smallest physical processing unit the mapping of the PE to actual hardware will change accordingly. Thus the PE layer separates the hardware specific part of the VM from the hardware-independent part. Each PE is labeled with an id number which identifies it uniquely within all of the VM instances of an ESMF application.
- Core: A Core is the smallest physical processing unit which typically comprises a register set, an integer arithmetic unit, a floating-point unit and various control units. Each Core is labeled with an id number which identifies it uniquely within all of the VM instances of an ESMF application.
- CPU: The central processing unit (CPU) houses single or multiple cores, providing them with the interface to system memory, interconnects and I/O. Typically the CPU provides some level of caching for the instruction and data streams in and out of the Cores. Cores in a multi-core CPU typically share some caches. Each CPU is labeled with an id number which identifies it uniquely within all of the VM instances of an ESMF application.
- SSI: A single system image (SSI) spans all the CPUs controlled by a single running instance of the operating system. SMP and NUMA are typical multi-CPU SSI architectures. Each SSI is labeled with an id number which identifies it uniquely within all of the VM instances of an ESMF application.
- TOE: A thread of execution (TOE) executes an instruction sequence. TOE's come in two flavors: PET and TET.

- PET: A persistent execution thread (PET) executes an instruction sequence on an associated set of data. The PET has a lifetime at least as long as the associated data set. In ESMF the PET is the central concept of abstraction provided by the VM class. The PETs of an VM object are labeled from 0 to N-1 where N is the total number of PETs in the VM object.
- TET: A transient execution thread (TET) executes an instruction sequence on an associated set of data. A TET's lifetime might be shorter than that of the associated data set.
- OS-Instance: The OS-Instance of a TOE describes how a particular TOE is instantiated on the OS level. Using POSIX terminology a TOE will run as a single thread within a single- or multi-threaded process.
- Pthreads: Communication via the POSIX Thread interface.
- MPI-1, MPI-2: Communication via MPI standards 1 and 2.
- armci: Communication via the aggregate remote memory copy interface.
- SHMEM: Communication via the SHMEM interface.
- OS-IPC: Communication via the operating system's inter process communication interface. Either POSIX IPC or System V IPC.
- InterCon-lib: Communication via the interconnect's library native interface. An example is the Elan library for Quadrics.

The POSIX machine abstraction, while a very powerful concept, needs augmentation when applied to HPC applications. Key elements of the POSIX abstraction are processes, which provide virtually unlimited resources (memory, I/O, sockets, ...) to possibly multiple threads of execution. Similarly POSIX threads create the illusion that there is virtually unlimited processing power available to each POSIX process. While the POSIX abstraction is very suitable for many multi-user/multi-tasking applications that need to share limited physical resources, it does not directly fit the HPC workload where over-subscription of resources is one of the most expensive modes of operation.

ESMF's virtual machine abstraction is based on the POSIX machine model but holds additional information about the available physical processing units in terms of Processing Elements (PEs). A PE is the smallest physical processing unit and encapsulates the hardware details (Cores, CPUs and SSIs).

There is exactly one physical machine layout for each application, and all VM instances have access to this information. The PE is the smallest processing unit which, in today's microprocessor technology, corresponds to a single Core. Cores are arranged in CPUs which in turn are arranged in SSIs. The setup of the physical machine layout is part of the ESMF initialization process.

On top of the PE concept the key abstraction provided by the VM is the PET. All user code is executed by PETs while OS and hardware details are hidden. The VM class contains a number of methods which allow the user to prescribe how the PETs of a desired virtual machine should be instantiated on the OS level and how they should map onto the hardware. This prescription is kept in a private virtual machine plan object which is created at the same time the associated component is being created. Each time component code is entered through one of the component's registered top–level methods (Initialize/Run/Finalize), the virtual machine plan along with a pointer to the respective user function is used to instantiate the user code on the PETs of the associated VM in form of single- or multi-threaded POSIX processes.

The process of starting, entering, exiting and shutting down a VM is very transparent, all spawning and joining of threads is handled by VM methods "behind the scenes". Furthermore, fundamental synchronization and communication primitives are provided on the PET level through a uniform API, hiding details related to the actual instantiation of the participating PETs.

Within a VM object each PE of the physical machine maps to 0 or 1 PETs. Allowing unassigned PEs provides a means to prevent over-subscription between multiple concurrently running virtual machines. Similarly a maximum of one PET per PE prevents over-subscription within a single VM instance. However, over-subscription is possible by subscribing PETs from different virtual machines to the same PE. This type of over-subscription can be desirable for PETs associated with I/O workloads expected to be used infrequently and to block often on I/O requests.

On the OS level each PET of a VM object is represented by a POSIX thread (Pthread) either belonging to a single– or multi–threaded process and maps to at least 1 PE of the physical machine, ensuring its execution. Mapping a single PET to multiple PEs provides resources for user–level multi–threading, in which case the user code inquires how many PEs are associated with its PET and if there are multiple PEs available the user code can spawn an equal number of threads (e.g. OpenMP) without risking over-subscription. Typically these user spawned threads are short-lived and used for fine-grained parallelization in form of TETs. All PEs mapped against a single PET must be part of a unique SSI in order to allow user–level multi–threading!

In addition to discovering the physical machine the ESMF initialization process sets up the default global virtual machine. This VM object, which is the ultimate parent of all VMs created during the course of execution, contains as many PETs as there are PEs in the physical machine. All of its PETs are instantiated in form of single-threaded MPI processes and a 1:1 mapping of PETs to PEs is used for the default global VM.

The VM design and implementation is based on the POSIX process and thread model as well as the MPI-1.2 standard. As a consequence of the latter standard the number of processes is static during the course of execution and is determined at start-up. The VM implementation further requires that the user starts up the ESMF application with as many MPI processes as there are PEs in the available physical machine using the platform dependent mechanism to ensure proper process placement.

All MPI processes participating in a VM are grouped together by means of an MPI_Group object and their context is defined via an MPI_Comm object (MPI intra-communicator). The PET local process id within each virtual machine is equal to the MPI_Comm_rank in the local MPI_Comm context whereas the PET process id is equal to the MPI_Comm_rank in MPI_COMM_WORLD. The PET process id is used within the VM methods to determine the virtual memory space a PET is operating in.

In order to provide a migration path for legacy MPI-applications the VM offers accessor functions to its MPI_Comm object. Once obtained this object may be used in explicit user-code MPI calls within the same context.

51.6 Class API

52 Profiling and Tracing

52.1 Description

52.1.1 Profiling

ESMF's built in *profiling* capability collects runtime statistics of an executing ESMF application through both automatic and manual code instrumentation. Timing information for all phases of all ESMF components executing in an application can be automatically collected using the `ESMF_RUNTIME_PROFILE` environment variable (see below for settings). Additionally, arbitrary user-defined code regions can be timed by manually instrumenting code with special API calls. Timing profiles of component phases and user-defined regions can be output in several different formats:

- in text at the end of ESMF Log files

- in separate text file, one per PET (if the ESMF Logs are turned off)
- in a single summary text file that aggregates timings over multiple PETs
- in a binary format for import into the esmf-profiler for profile visualization

The following table lists important environment variables that control aspects of ESMF profiling.

Environment Variable	Description	Example Values
ESMF_RUNTIME_PROFILE	Enable/disables all profiling functions	ON or OFF
ESMF_RUNTIME_PROFILE_PETLIST	Limits profiling to an explicit list of PETs	"0-9 50 99"
ESMF_RUNTIME_PROFILE_OUTPUT	Controls output format of profiles; multiple can be specified in a space separated list	TEXT, SUMMARY, BINARY

52.1.2 Tracing

Whereas profiling collects summary information from an application, *tracing* records a more detailed set of events for later analysis. Trace analysis can be used to understand what happened during a program's execution and is often used for diagnosing problems, debugging, and performance analysis.

ESMF has a built-in tracing capability that records events into special binary log files. Unlike log files written by the ESMF_Log class, which are primarily for human consumption (see Section 49.1), the trace output files are recorded in a compact binary representation and are processed by tools to produce various analyses. ESMF event streams are recorded in the Common Trace Format (CTF). CTF traces include one or more event streams, as well as a metadata file describing the events in the streams.

Several tools are available for reading in the CTF traces output by ESMF. Of the tools listed below, the first one is designed specifically for analyzing ESMF applications and the second two are general purpose tools for working with all CTF traces.

- esmf-profiler is a tool that ingests traces from an ESMF application and generates performance profile plots.
- TraceCompass is a general purpose tool for reading, analyzing, and visualizing traces.
- Babeltrace is a command-line tool and library for trace conversion that can read and write CTF traces. Python bindings are available to open CTF traces are iterate through events.

Events that can be captured by the ESMF tracer include the following. Events are recorded with a high-precision timestamp to allow timing analyses.

phase_enter indicates entry into an initialize, run, or finalize ESMF component routine

phase_exit indicates exit from an initialize, run, or finalize ESMF component routine

region_enter indicates entry into a user-defined code region

region_exit indicates exit from a user-defined code region

The following table lists important environment variables that control aspects of ESMF tracing.

Environment Variable	Description	Example Values
ESMF_RUNTIME_TRACE	Enable/disables all tracing functions	ON or OFF
ESMF_RUNTIME_TRACE_CLOCK	Sets the type of clock for timestamping events (see Section 52.2.6).	REALTIME or MONO MONOTONIC_SYNC
ESMF_RUNTIME_TRACE_PETLIST	Limits tracing to an explicit list of PETs	"0-9 50 99"
ESMF_RUNTIME_TRACE_COMPONENT	Enables/disable tracing of Component phase_enter and phase_exit events	ON or OFF
ESMF_RUNTIME_TRACE_FLUSH	Controls frequency of event stream flushing to file	DEFAULT or EAGER

52.2 Use and Examples

52.2.1 Output a Timing Profile to Text

ESMF profiling is disabled by default. To profile an application, set the ESMF_RUNTIME_PROFILE variable to ON prior to executing the application. You do not need to recompile your code to enable profiling.

```
# csh shell
$ setenv ESMF_RUNTIME_PROFILE ON

# bash shell
$ export ESMF_RUNTIME_PROFILE=ON

# (from now on, only the csh shell version will be shown)
```

Then execute the application in the usual way. At the end of the run the profile information will be available at the end of each PET log (if ESMF Logs are turned on) or in a set of separate files, one per PET, with names *ESMF_Profile.XXX* where XXX is the PET number. Below is an example timing profile. Some regions are left out for brevity.

```
Region                          Count  Total (s)   Self (s)    Mean (s)    Min (s)     Max
  [esm] Init 1                  1      4.0878      0.0341      4.0878      4.0878      4.0
    [OCN-TO-ATM] IPDv05p6b      1      2.6007      2.6007      2.6007      2.6007      2.6
    [ATM-TO-OCN] IPDv05p6b      1      1.4333      1.4333      1.4333      1.4333      1.4
    [ATM] IPDv00p2              1      0.0055      0.0055      0.0055      0.0055      0.0
    [OCN] IPDv00p2              1      0.0023      0.0023      0.0023      0.0023      0.0
    [ATM] IPDv00p1              1      0.0011      0.0011      0.0011      0.0011      0.0
    [OCN] IPDv00p1              1      0.0009      0.0009      0.0009      0.0009      0.0
    [ATM-TO-OCN] IPDv05p3       1      0.0008      0.0008      0.0008      0.0008      0.0
    [ATM-TO-OCN] IPDv05p1       1      0.0008      0.0008      0.0008      0.0008      0.0
    [ATM-TO-OCN] IPDv05p2b      1      0.0007      0.0007      0.0007      0.0007      0.0
    [ATM-TO-OCN] IPDv05p4       1      0.0007      0.0007      0.0007      0.0007      0.0
    [ATM-TO-OCN] IPDv05p2a      1      0.0007      0.0007      0.0007      0.0007      0.0
    [ATM-TO-OCN] IPDv05p5       1      0.0007      0.0007      0.0007      0.0007      0.0
    [OCN-TO-ATM] IPDv05p3       1      0.0006      0.0006      0.0006      0.0006      0.0
    [OCN-TO-ATM] IPDv05p4       1      0.0006      0.0006      0.0006      0.0006      0.0
    [OCN-TO-ATM] IPDv05p2b      1      0.0006      0.0006      0.0006      0.0006      0.0
    [OCN-TO-ATM] IPDv05p2a      1      0.0006      0.0006      0.0006      0.0006      0.0
    [OCN-TO-ATM] IPDv05p5       1      0.0006      0.0006      0.0006      0.0006      0.0
    [OCN-TO-ATM] IPDv05p1       1      0.0005      0.0005      0.0005      0.0005      0.0
```

```
  [esm] RunPhase1                 1      2.7423      0.9432      2.7423      2.7423      2.7
    [OCN-TO-ATM] RunPhase1        864    0.6094      0.6094      0.0007      0.0006      0.0
    [ATM] RunPhase1               864    0.5296      0.2274      0.0006      0.0005      0.0
      ATM:ModelAdvance            864    0.3022      0.3022      0.0003      0.0003      0.0
    [ATM-TO-OCN] RunPhase1        864    0.3345      0.3345      0.0004      0.0002      0.0
    [OCN] RunPhase1               864    0.3256      0.3256      0.0004      0.0003      0.0
  [esm] FinalizePhase1            1      0.0029      0.0020      0.0029      0.0029      0.0
    [OCN-TO-ATM] FinalizePhase1   1      0.0006      0.0006      0.0006      0.0006      0.0
    [ATM-TO-OCN] FinalizePhase1   1      0.0002      0.0002      0.0002      0.0002      0.0
    [OCN] FinalizePhase1          1      0.0001      0.0001      0.0001      0.0001      0.0
    [ATM] FinalizePhase1          1      0.0000      0.0000      0.0000      0.0000      0.0
```

A timed region is either an ESMF component phase (e.g., initialize, run, or finalize) or a user-defined region of code surrounded by calls to `ESMF_TraceRegionEnter()` and `ESMF_TraceRegionExit()`. (See section **??** for more information on instrumenting user-defined regions.) Regions are organized hierarchically with sub-regions nested. For example, in the profile above, the `[OCN] RunPhase1` is a sub-region of `[esm] RunPhase1` and is entirely contained inside that region. Regions with the same name may appear at multiple places in the hierarchy, and so would appear in multiple rows in the table. The statistics in that row apply to that region at that location in the hierarchy. Component names appear in square brackets, e.g., `[ATM]`, `[OCN]`, and `[ATM-TO-OCN]`. By default, timings are based on elapsed wall clock time and are collected on a per-PET basis. Therefore, regions timings may differ across PETs. Regions are sorted with the most expensive regions appearing at the top. The following describes the meaning of the statistics in each column:

`Count` the number of times the region is executed

`Total` the aggregate time spent in the region, inclusive of all sub-regions

`Self` the aggregate time spend in the region, exclusive of all sub-regions

`Mean` the average amount of time for one execution of the region

`Min` time of the fastest execution of the region

`Max` time of the slowest execution of the region

52.2.2 Summarize Timings across Multiple PETs

By default, separate timing profiles are generated for each PET in the application. The per-PET profiles can be aggregated together and output to a single file, *ESMF_Profile.summary*, by setting the `ESMF_RUNTIME_PROFILE_OUTPUT` environment variable as follows:

```
$ setenv ESMF_RUNTIME_PROFILE ON              # turn on profiling
$ setenv ESMF_RUNTIME_PROFILE_OUTPUT SUMMARY  # specify summary output
```

Note the `ESMF_RUNTIME_PROFILE` environment variable must also be set to `ON` since this controls all profiling capabilities. The *ESMF_Profile.summary* file will contain a tree of timed regions, but aggregated across all PETs. For example:

```
Region                          PETs   PEs    Count    Mean (s)    Min (s)     Min PET Max
  [esm] Init 1                  4      4      1        4.0880      4.0878      2       4.0
```

```
  [OCN-TO-ATM] IPDv05p6b       4      4      1        2.6007      2.6007      2       2.6
  [ATM-TO-OCN] IPDv05p6b       4      4      1        1.4335      1.4333      0       1.4
  [ATM-TO-OCN] IPDv05p4        4      4      1        0.0037      0.0007      0       0.0
  [ATM] IPDv00p2               4      4      1        0.0034      0.0020      1       0.0
  [ATM-TO-OCN] IPDv05p1        4      4      1        0.0020      0.0007      2       0.0
  [OCN] IPDv00p2               4      4      1        0.0019      0.0015      3       0.0
  [ATM-TO-OCN] IPDv05p3        4      4      1        0.0010      0.0008      0       0.0
  [ATM-TO-OCN] IPDv05p2a       4      4      1        0.0009      0.0007      0       0.0
  [ATM] IPDv00p1               4      4      1        0.0009      0.0007      3       0.0
  [ATM-TO-OCN] IPDv05p2b       4      4      1        0.0008      0.0007      0       0.0
  [ATM-TO-OCN] IPDv05p5        4      4      1        0.0008      0.0007      0       0.0
  [ATM-TO-OCN] IPDv05p6a       4      4      1        0.0008      0.0005      2       0.0
  [OCN-TO-ATM] IPDv05p3        4      4      1        0.0008      0.0006      2       0.0
  [OCN-TO-ATM] IPDv05p4        4      4      1        0.0008      0.0006      0       0.0
  [OCN-TO-ATM] IPDv05p2b       4      4      1        0.0007      0.0006      2       0.0
  [OCN] IPDv00p1               4      4      1        0.0007      0.0005      1       0.0
  [OCN-TO-ATM] IPDv05p2a       4      4      1        0.0007      0.0006      2       0.0
  [OCN-TO-ATM] IPDv05p5        4      4      1        0.0007      0.0006      0       0.0
  [OCN-TO-ATM] IPDv05p1        4      4      1        0.0006      0.0005      0       0.0
  [OCN-TO-ATM] IPDv05p6a       4      4      1        0.0006      0.0004      2       0.0
[esm] RunPhase1                4      4      1        2.7444      2.7423      0       2.7
  [OCN-TO-ATM] RunPhase1       4      4      864      0.6123      0.6004      2       0.6
  [ATM] RunPhase1              4      4      864      0.5386      0.5296      0       0.5
    ATM:ModelAdvance           4      4      864      0.3038      0.3022      0       0.3
  [OCN] RunPhase1              4      4      864      0.3471      0.3256      0       0.3
  [ATM-TO-OCN] RunPhase1       4      4      864      0.2843      0.1956      1       0.3
[esm] FinalizePhase1           4      4      1        0.0029      0.0029      1       0.0
  [OCN-TO-ATM] FinalizePhase1  4      4      1        0.0007      0.0006      0       0.0
  [ATM-TO-OCN] FinalizePhase1  4      4      1        0.0002      0.0001      3       0.0
  [OCN] FinalizePhase1         4      4      1        0.0001      0.0001      3       0.0
  [ATM] FinalizePhase1         4      4      1        0.0001      0.0000      0       0.0
```

The meaning of the statistics in each column in as follows:

`PETs` the number of reporting PETs that executed the region

`PEs` the number of PEs associated with the reporting PETs that executed the region

`Count` the number of times each reporting PET executed the region or "MULTIPLE" if not all PETs executed the region the same number of times

`Mean` the mean across all reporting PETs of the total time spent in the region

`Min` the minimum across all reporting PETs of the total time spent in the region

`Min PET` the PET that reported the minimum time

`Max` the maximum across all reporting PETs of the total time spent in the region

`Max PET` the PET that reported the maximum time

Note that setting the `ESMF_RUNTIME_PROFILE_PETLIST` environment variable (described below) may reduce the number of reporting PETs. Only reporting PETs are included in the summary profile. To output both the per-PET and summary timing profiles, set the `ESMF_RUNTIME_PROFILE_OUTPUT` environment variable as follows:

```
$ setenv ESMF_RUNTIME_PROFILE_OUTPUT "TEXT SUMMARY"
```

52.2.3 Limit the Set of Profiled PETs

By default, all PETs in an application are profiled. It may be desirable to only profile a subset of PETs to reduce the amount of output. An explicit list of PETs can be specified by setting the `ESMF_RUNTIME_PROFILE_PETLIST` environment variable. The syntax of this environment variable is to list PET numbers separated by spaces. PET ranges are also supported using the "X-Y" syntax where X < Y. For example:

```
# only profile PETs 0, 20, and 35 through 39
$ setenv ESMF_RUNTIME_PROFILE_PETLIST "0 20 35-39"
```

When used in conjunction with the `SUMMARY` option above, the summarized profile will only aggregate over the specified set of PETs. The one exception is that PET 0 is always profiled if `ESMF_RUNTIME_PROFILE=ON`, regardless of the `ESMF_RUNTIME_TRACE_PETLIST` setting.

52.2.4 Include MPI Communication in the Profile

MPI functions can be included in the timing profile to indicate how much time is spent inside communication calls. This can also help to determine load imbalance in the system, since large times spent inside MPI may indicate that communication between PETs is not tightly synchronized. This option includes *all* MPI calls in the application, whether or not they originate from the ESMF library. Here is a partial example summary profile that contains MPI times:

```
Region                          PETs   Count    Mean (s)     Min (s)      Min PET Max (s)
  [esm] RunPhase1               8      1        4.9307       4.6867       0       4.9656
    [OCN] RunPhase1             8      1824     0.8344       0.8164       0       0.8652
    [MED] RunPhase1             8      1824     0.8203       0.7900       5       0.8584
    [ATM] RunPhase1             8      1824     0.6387       0.6212       5       0.6610
    [ATM-TO-MED] RunPhase1      8      1824     0.5975       0.5317       0       0.6583
      MPI_Bcast                 8      1824     0.0443       0.0025       4       0.1231
      MPI_Wait                  8      MULTIPLE 0.0421       0.0032       0       0.0998
    [MED-TO-OCN] RunPhase1      8      1824     0.4879       0.4497       0       0.5362
      MPI_Wait                  8      MULTIPLE 0.0234       0.0030       0       0.0821
      MPI_Bcast                 8      1824     0.0111       0.0024       4       0.0273
    [OCN-TO-MED] RunPhase1      8      1824     0.4541       0.4075       0       0.4918
      MPI_Wait                  8      MULTIPLE 0.0339       0.0017       0       0.0824
      MPI_Bcast                 8      1824     0.0194       0.0026       4       0.0452
    [MED-TO-ATM] RunPhase1      8      1824     0.4487       0.4005       0       0.4911
      MPI_Bcast                 8      1824     0.0338       0.0026       4       0.0942
      MPI_Wait                  8      MULTIPLE 0.0241       0.0022       1       0.0817
  [esm] Init 1                  8      1        0.6287       0.6287       1       0.6287
    [ATM-TO-MED] IPDv05p6b      8      1        0.1501       0.1500       1       0.1501
      MPI_Barrier               8      242      0.0082       0.0006       3       0.0157
      MPI_Wait                  8      MULTIPLE 0.0034       0.0010       0       0.0053
      MPI_Allreduce             8      62       0.0030       0.0003       3       0.0063
      MPI_Alltoall              8      6        0.0015       0.0000       1       0.0022
      MPI_Allgather             8      21       0.0010       0.0002       1       0.0017
```

```
      MPI_Waitall               8       MULTIPLE 0.0006      0.0001      3       0.0015
      MPI_Send                  8       MULTIPLE 0.0004      0.0001      7       0.0008
      MPI_Allgatherv            8       6        0.0001      0.0001      4       0.0001
      MPI_Scatter               8       5        0.0000      0.0000      0       0.0000
      MPI_Reduce                8       5        0.0000      0.0000      1       0.0000
      MPI_Recv                  8       MULTIPLE 0.0000      0.0000      0       0.0000
      MPI_Bcast                 8       1        0.0000      0.0000      0       0.0000
```

The procedure for including MPI functions in the timing profile depends on whether the application is dynamically or statically linked. Most applications are dynamically linked, however on some systems (such as Cray), static linking may be used. Note that for either option, ESMF must be built with `ESMF_TRACE_LIB_BUILD=ON`, which is the default.

In *dynamically linked applications*, the `LD_PRELOAD` (Linux) or `DYLD_INSERT_LIBRARIES` (Darwin) environment variable must be used when executing the MPI application. This instructs the dynamic loader to interpose certain MPI symbols so they can be captured by the ESMF profiler. To simplify this process, a script is provided at `$(ESMF_INSTALL_LIBDIR)/preload.sh` that sets the appropriate variable.

For example, if you typically execute your application as as follows:

```
$ mpirun -np 8 ./myApp
```

then you should add the *preload.sh* script in front of the executable when starting the application as follows:

```
# replace $(ESMF_INSTALL_LIBDIR) with absolute path
# ... to the ESMF installation lib directory
$ mpirun -np 8 $(ESMF_INSTALL_LIBDIR)/preload.sh ./myApp
```

An advantage of this approach is that your application does *not* need to be recompiled. The MPI timing information will be included in the per-PET profiles and/or the summary profile, depending on the setting of environment variable `ESMF_RUNTIME_PROFILE_OUTPUT`.

Notice that an additional step is required for dynamically linked applications on *Darwin* systems with System Integrity Protection (SIP) enabled! In addition to using the `$(ESMF_INSTALL_LIBDIR)/preload.sh` script during launching of the executable as shown above, the executable must *also be linked* against the dynamic ESMF trace preload library. This must happen during the link step of the executable. It is most easily accomplished by using variable `$(ESMF_F90ESMFPRELOADLINKLIBS)` instead of the typical `$(ESMF_F90ESMFLINKLIBS)` variable for the final link command. Both variables are defined in the *esmf.mk* file that should be imported by the application Makefile. For example:

```
# import esmf.mk
include $(ESMFMKFILE)

# other makefile targets here...

# example final link command, with $(ESMF_F90ESMFPRELOADLINKLIBS)
myApp: myApp.o driver.o model.o
        $(ESMF_F90LINKER) $(ESMF_F90LINKOPTS) $(ESMF_F90LINKPATHS) \
        $(ESMF_F90LINKRPATHS) -o $@ $^ $(ESMF_F90ESMFPRELOADLINKLIBS)
```

In *statically linked applications*, the application must be re-linked with specific options provided to the linker. These options instruct the linker to wrap the MPI symbols with the ESMF profiling functions. The linking flags that must be provided are included in the *esmf.mk* Makefile fragment that is part of the ESMF installation. These link flags should be imported into your application Makefile, and included in the final link command. To do this, first import the *esmf.mk* file into your application Makefile. The path to this file is typically stored in the ESMFMKFILE environment variable. Then, pass the variables $(ESMF_TRACE_STATICLINKOPTS) and $(ESMF_TRACE_STATICLINKLIBS) to the final linking command. For example:

```
# import esmf.mk
include $(ESMFMKFILE)

# other makefile targets here...

# example final link command, with $(ESMF_TRACE_STATICLINKOPTS)
# ... and $(ESMF_TRACE_STATICLINKLIBS) added
myApp: myApp.o driver.o model.o
        $(ESMF_F90LINKER) $(ESMF_F90LINKOPTS) $(ESMF_F90LINKPATHS) \
        $(ESMF_F90LINKRPATHS) -o $@ $^ $(ESMF_F90ESMFLINKLIBS) \
        $(ESMF_TRACE_STATICLINKOPTS) $(ESMF_TRACE_STATICLINKLIBS)
```

This option will statically wrap all of the MPI functions and include them in the profile output. Execute the application in the normal way with the environment variable ESMF_RUNTIME_PROFILE set to ON. You will see the MPI functions included in the timing profile.

52.2.5 Output a Detailed Trace for Analysis

ESMF tracing is disabled by default. To enable tracing, set the ESMF_RUNTIME_TRACE environment variable to ON. You do not need to recompile your code to enable tracing.

```
# csh shell
$ setenv ESMF_RUNTIME_TRACE ON

# bash shell
$ export ESMF_RUNTIME_TRACE=ON
```

When enabled, the default behavior is to trace all PETs of the ESMF application. Although the ESMF tracer is designed to write events in a compact form, tracing can produce an extremely large number of events depending on the total number of PETs and the length of the run. To reduce output, it is possible to restrict the PETs that produce trace output by setting the ESMF_RUNTIME_TRACE_PETLIST environment variable. For example, this setting:

```
$ setenv ESMF_RUNTIME_TRACE_PETLIST "0 101 192-196"
```

will instruct the tracer to only trace PETs 0, 101, and 192 through 196 (inclusive). The syntax of this environment variable is to list PET numbers separated by spaces. PET ranges are also supported using the "X-Y" syntax where X < Y. For PET counts greater than 100, it is recommended to set this environment variable. The one exception is that PET 0 is always traced, regardless of the ESMF_RUNTIME_TRACE_PETLIST setting.

ESMF's profiling and tracing options can be used together. A typical use would be to set ESMF_RUNTIME_PROFILE=ON for all PETs to capture summary timings, and set ESMF_RUNTIME_TRACE=ON

and ESMF_RUNTIME_TRACE_PETLIST to a subset of of PETs, such as the root PET of each ESMF component. This helps to keep trace sizes small while still providing timing summaries over all PETs.

When tracing is enabled, phase_enter and phase_exit events will automatically be recorded for all initialize, run, and finalize phases of all Components in the application. To trace *only* user-instrumented regions (via the ESMF_TraceRegionEnter() and ESMF_TraceRegionExit() calls), Component-level tracing can be turned off by setting:

```
$ setenv ESMF_RUNTIME_TRACE_COMPONENT OFF
```

After running an ESMF application with tracing enabled, a directory called *traceout* will be created in the run directory and it will contain a *metadata* file and an event stream file *esmf_stream_XXXX* for each PET with tracing enabled. Together these files form a valid CTF trace which may be analyzed with any of the tools listed above.

Trace events are flushed to file at a regular interval. If the application crashes, some of the most recent events may not be flushed to file. To maximize the number of events appearing in the trace, an option is available to flush events to file more frequently. Because this option may have negative performance implications due to increased file I/O, it is not recommended unless needed. To turn on eager flushing use:

```
$ setenv ESMF_RUNTIME_TRACE_FLUSH EAGER
```

52.2.6 Set the Clock used for Profiling/Tracing

There are three options for the kind of clock to use to timestamp events when profiling/tracing an application. These options are controlled by setting the environment variable ESMF_RUNTIME_TRACE_CLOCK.

REALTIME The REALTIME clock timestamps events with the current time on the system. This is the default clock if the above environment variable is not set. This setting can be useful when tracing PETs that span multiple physical computing nodes assuming that the system clocks on each node are adequately synchronized. On most HPC systems, system clocks are periodically updated to stay in sync. A disadvantage of this clock is that periodic adjustments mean the clock is not monotonically increasing so some timings may be inaccurate if the system clock jumps forward or backward significantly. Testing has shown that this is not typically an issue on most systems.

MONOTONIC The MONOTONIC clock is guaranteed to be monotonically increasing and does not suffer from periodic adjustments. The timestamps represent an amount of time since some arbitrary point in the past. There is no guarantee that these timestamps will be synchronized across physical computing nodes, so this option should only be used for tracing a set of PETs running on a single physical machine.

MONOTONIC_SYNC The MONOTONIC_SYNC clock is similar to the MONOTONIC clock in that it is guaranteed to be monotonically increasing. In addition, at application startup, all PET clocks are synchronized to a common time by determining a PET-local offset to be applied to timestamps. Therefore this option can be used to compare trace streams across physical nodes.

52.3 Restrictions and Future Work

1. **Limited types of trace events.** Currently only a few trace event types are available. The tracer may be extended in the future to record additional types of events.

2. **MPI call profing not available for statically linked executables on Darwin.** Currently the linker on Darwin systems does not support the wrapping of symbols during static linking. In order to access MPI call profiling on Darwin, executables should be linked dynamically in combination with the procedure described in section 52.2.4.

52.4 Class API

53 Fortran I/O and System Utilities

53.1 Description

The ESMF Fortran I/O and System utilities provide portable methods to access capabilities which are often implemented in different ways amongst different environments. These utility methods are divided into three groups: command line access, Fortran I/O, and sorting.

Command line arguments may be accessed using three methods: `ESMF_UtilGetArg()` returns a given command line argument, `ESMF_UtilGetArgC()` returns a count of the number of command line arguments available. Finally, the `ESMF_UtilGetArgIndex()` method returns the index of a desired argument value, given its keyword name.

Two I/O methods are implemented: `ESMF_IOUnitGet()`, to obtain an unopened Fortran unit number within the range of unit numbers that ESMF is allowed to use, and `ESMF_IOUnitFlush()` to flush the I/O buffer associated with a specific Fortran unit.

Finally, the `ESMF_UtilSort()` method sorts integer, floating point, and character string data types in either ascending or descending order.

53.2 Use and Examples

53.2.1 Fortran unit number management

The `ESMF_UtilIOUnitGet()` method is provided so that applications using ESMF can remain free of unit number conflicts — both when combined with other third party code, or with ESMF itself. This call is typically used just prior to an `OPEN` statement:

```
call ESMF_UtilIOUnitGet (unit=grid_unit, rc=rc)
open (unit=grid_unit, file='grid_data.dat', status='old', action='read')
```

By default, unit numbers between 50 and 99 are scanned to find an unopened unit number.

Internally, ESMF also uses `ESMF_UtilIOUnitGet()` when it needs to open Fortran unit numbers for file I/O. By using the same API for both user and ESMF code, unit number collisions can be avoided.

When integrating ESMF into an application where there are conflicts with other uses of the same unit number range, such as when hard-coded unit number values are used, an alternative unit number range can be specified. The `ESMF_Initialize()` optional arguments `IOUnitLower` and `IOUnitUpper` may be set as needed. Note that `IOUnitUpper` must be set to a value higher than `IOUnitLower`, and that both must be non-negative. Otherwise `ESMF_Initialize` will return a return code of `ESMF_FAILURE`. ESMF itself does not typically need more than about five units for internal use.

```
call ESMF_Initialize (..., IOUnitLower=120, IOUnitUpper=140)
```

All current Fortran environments have preconnected unit numbers, such as units 5 and 6 for standard input and output, in the single digit range. So it is recommended that the unit number range is chosen to begin at unit 10 or higher to avoid these preconnected units.

53.2.2 Flushing output

Fortran run-time libraries generally use buffering techniques to improve I/O performance. However output buffering can be problematic when output is needed, but is "trapped" in the buffer because it is not full. This is a common occurrance when debugging a program, and inserting `WRITE` statements to track down the bad area of code. If the program crashes before the output buffer has been flushed, the desired debugging output may never be seen — giving a misleading indication of where the problem occurred. It would be desirable to ensure that the output buffer is flushed at predictable points in the program in order to get the needed results. Likewise, in parallel code, predictable flushing of output buffers is a common requirement, often in conjunction with `ESMF_VMBarrier()` calls.

The `ESMF_UtilIOUnitFlush()` API is provided to flush a unit as desired. Here is an example of code which prints debug values, and serializes the output to a terminal in PET order:

```
type(ESMF_VM) :: vm

integer :: tty_unit
integer :: me, npets

call ESMF_Initialize (vm=vm, rc=rc)
call ESMF_VMGet (vm, localPet=me, petCount=npes)

call ESMF_UtilIOUnitGet (unit=tty_unit)
open (unit=tty_unit, file='/dev/tty', status='old', action='write')
...
call ESMF_VMBarrier (vm=vm)
do, i=0, npets-1
  if (i == me) then
    write (tty_unit, *) 'PET: ', i, ', values are: ', a, b, c
    call ESMF_UtilIOUnitFlush (unit=tty_unit)
  end if
  call ESMF_VMBarrier (vm=vm)
end do
```

53.3 Design and Implementation Notes

53.3.1 Fortran unit number management

When ESMF needs to open a Fortran I/O unit, it calls `ESMF_IOUnitGet()` to find an unopened unit number. As delivered, the range of unit numbers that are searched are between `ESMF_LOG_FORTRAN_UNIT_NUMBER` (normally set to 50), and `ESMF_LOG_UPPER` (normally set to 99.) Unopened unit numbers are found by using the Fortran `INQUIRE` statement.

When integrating ESMF into an application where there are conflicts with other uses of the same unit number range, an alternative range can be specified in the `ESMF_Initialize()` call by setting the `IOUnitLower` and

IOUnitUpper arguments as needed. ESMF_IOUnitGet() will then search the alternate range of unit numbers. Note that IOUnitUpper must be set to a value higher than IOUnitLower, and that both must be non-negative. Otherwise ESMF_Initialize will return a return code of ESMF_FAILURE.

Fortran unit numbers are not standardized in the Fortran 90 Standard. The standard only requires that they be non-negative integers. But other than that, it is up to the compiler writers and application developers to provide and use units which work with the particular implementation. For example, units 5 and 6 are a defacto standard for "standard input" and "standard output" — even though this is not specified in the actual Fortran standard. The Fortran standard also does not specify which unit numbers can be used, nor does it specify how many can be open simultaneously.

Since all current compilers have preconnected unit numbers, and these are typically found on units lower than 10, it is recommended that applications use unit numbers 10 and higher.

53.3.2 Flushing output

When ESMF needs to flush a Fortran unit, the ESMF_IOUnitFlush() API is used to centralize the file flushing capability, because Fortran has not historically had a standard mechanism for flushing output buffers. Most compilers run-time libraries support various library extensions to provide this functionality — though, being non-standard, the spelling and number of arguments vary between implementations. Fortran 2003 also provides for a FLUSH statement which is built into the language. When possible, ESMF_IOUnitFlush() uses the F2003 FLUSH statement. With older compilers, the appropriate library call is made.

53.3.3 Sorting algorithms

The ESMF_UtilSort() algorithms are the same as those in the LAPACK sorting procedures SLASRT() and DLASRT(). Two algorithms are used. For small sorts, arrays with 20 or fewer elements, a simple Insertion sort is used. For larger sorts, a Quicksort algorithm is used.

Compared to the original LAPACK code, a full Fortran 90 style interface is supported for ease of use and enhanced compile time checking. Additional support is also provided for integer and character string data types.

53.4 Utility API

Part VI

References

References

[1] *JSON for Modern C++*, 2020 (accessed February 2020). https://github.com/nlohmann/json.

[2] *JSON for Modern C++ 64-Bit Float*, 2020 (accessed February 2020). https://nlohmann.github.io/json/classnlohmann_1_1basic__json_a88d6103cb3620410b35200ee8

[3] *JSON for Modern C++ 64-Bit Integer*, 2020 (accessed February 2020). https://nlohmann.github.io/json/classnlohmann_1_1basic__json_a98e611d67b7bd75307de99c93

[4] *JSON for Modern C++ JSON Pointer*, 2020 (accessed February 2020). https://nlohmann.github.io/json/classnlohmann_1_1json__pointer_a7f32d7c62841f0c4a6784cf

[5] *JSON for Modern C++ Memory Efficiency*, 2020 (accessed February 2020). https://github.com/nlohmann/json#design-goals.

[6] *JSON*, 2020 (accessed March 2020). https://www.json.org/json-en.html.

[7] SCRIP: A Spherical Coordinate Remapping and Interpolation Package. http://oceans11.lanl.gov/trac/SCRIP, last accessed on Dec 4, 2015. Los Alamos Software Release LACC 98-45.

[8] NetCDF Climate and Forecast (CF) Metadata Conventions. http://cfconventions.org/, last accessed on Nov 27, 2015.

[9] NetCDF Users Guide for C, Version 3. http://www.unidata.ucar.edu/software/netcdf/docs, last accessed on Nov 27, 2015.

[10] V. Balaji, Jeff Anderson, Isaac Held, Michael Winton, Jeff Durachta, Sergey Malyshev, and Ronald J. Stouffer. The exchange grid: a mechanism for data exchange between earth system components on independent grids. *Parallel Computational Fluid Dynamics: Theory and Applications, Proceedings of the 2005 International Conference on Parallel Computational Fluid Dynamics*, 2006.

[11] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring. *Scientific Programming*, 20(2), 2012.

[12] Y. Meurdesoif E. Kritsikis, M. Aechtner and T. Dubos. Conservative interpolation between general spherical meshes. *Geoscientific Model Development*, 10, 2017.

[13] H. C. Edwards, A. B. Williams, G. D. Sjaardema, D. G. Baur, and W. K. Cochran. SIERRA toolkit computational mesh conceptual model. Technical Report SAND2010-1192, Sandia National Laboratories, Albuquerque, New Mexico 87185, March 2010.

[14] H. Gu, Z. Zong, and K.C. Hung. A modified superconvergent patch recovery method and its application to large deformation problems. *Finite Elements in Analysis and Design*, 40(5-6), 2004.

[15] International Organization for Standardization. Standard 8601:2004, Data elements and interchange formats – Information interchange – Representation of dates and times. http://www.iso.ch/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=40874&COMMID=&scopelist=, last accessed on Dec 4, 2015.

[16] A.R. Khoei and S.A. Gharehbaghi. The superconvergent patch recovery technique and data transfer operators in 3d plasticity problems. *Finite Elements in Analysis and Design*, 43(8), 2007.

[17] Peter Meyer. A good discussion of Gregorian and Julian Calendars. http://www.hermetic.ch/cal_stud/cal_art.html, last accessed on Nov 27, 2015.

[18] D. Ramshaw. Conservative rezoning algorithm for generalized two-dimensional meshes. *Journal of Computational Physics*, 59, 1985.

[19] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[20] Seidelman, P.K. *Explanatory Supplement to the Astronomical Almanac*. University Science Books, 1992.

Part VII

Appendices

54 Appendix A: Master List of Constants

54.1 ESMF_ALARMLIST

This flag is documented in section 46.2.1.

54.2 ESMF_DIM_ARB

DESCRIPTION:
An integer named constant which is used to indicate that a particular dimension is arbitrarily distributed.

54.3 ESMF_ATTCOPY

This flag is documented in section 57.1.1.

54.4 ESMF_ATTGETCOUNT

This flag is documented in section 57.1.2.

54.5 ESMF_ATTNEST

DESCRIPTION:
Indicate whether or not to descend the Attribute hierarchy.

The type of this flag is:

`type(ESMF_AttNest_Flag)`

The valid values are:

ESMF_ATTNEST_ON Indicates that the Attribute hierarchy should be traversed.

ESMF_ATTNEST_OFF Indicates that the Attribute hierarchy should not be traversed.

54.6 ESMF_ATTRECONCILE

DESCRIPTION:
Indicate whether or not to handle metadata (Attributes) in `ESMF_StateReconcile()`.

The type of this flag is:

`type(ESMF_AttReconcileFlag)`

The valid values are:

ESMF_ATTRECONCILE_ON Attribute reconciliation will be turned on.

ESMF_ATTRECONCILE_OFF Attribute reconciliation will be turned off.

54.7 ESMF_ATTWRITE

This flag is documented in section 57.1.3.

54.8 ESMF_CALKIND

This flag is documented in section 42.2.1.

54.9 ESMF_COMPTYPE

DESCRIPTION:
Indicate the type of a Component.

The type of this flag is:

`type(ESMF_CompType_Flag)`

The valid values are:

ESMF_COMPTYPE_GRID A `ESMF_GridComp` object.

ESMF_COMPTYPE_CPL A `ESMF_CplComp` objects.

ESMF_COMPTYPE_SCI A `ESMF_SciComp` objects.

54.10 ESMF_CONTEXT

DESCRIPTION:
Indicates the type of VM context in which a Component will be executing its standard methods.

The type of this flag is:

`type(ESMF_Context_Flag)`

The valid values are:

ESMF_CONTEXT_OWN_VM The component is running in its own, separate VM context. Resources are inherited from the parent but can be arranged to fit the component's requirements.

ESMF_CONTEXT_PARENT_VM The component uses the parent's VM for resource management. Compared to components that use their own VM context components that run in the parent's VM context are more light-weight with respect to the overhead of calling into their initialize, run and finalize methods. Furthermore, VM-specific properties remain unchanged when going from the parent component to the child component. These properties include the MPI communicator, the number of PETs, the PET labeling, communication attributes, threading-level.

54.11 ESMF_COORDSYS

DESCRIPTION:
A set of values which indicates in which system the coordinates in a class (e.g. Grid) are. This type is useful both to indicate to other users the type of the coordinates, but also to control how the coordinates are interpreted in ESMF methods which depend on the coordinates (e.g. regridding methods like `ESMF_FieldRegridStore()`).

The type of this flag is:

`type(ESMF_CoordSys_Flag)`

The valid values are:

ESMF_COORDSYS_CART Cartesian coordinate system. In this system, the Cartesian coordinates are mapped to the coordinate dimensions in the following order: x,y,z. (E.g. using `coordDim=2` in ESMF_GridGetCoord() references the y dimension)

ESMF_COORDSYS_SPH_DEG Spherical coordinates in degrees. In this system, the spherical coordinates are mapped to the coordinate dimensions in the following order: longitude, latitude, radius. (E.g. using `coordDim=2` in ESMF_GridGetCoord() references the latitude dimension.)

ESMF_COORDSYS_SPH_RAD Spherical coordinates in radians. In this system, the spherical coordinates are mapped to the coordinate dimensions in the following order: longitude, latitude, radius. (E.g. using `coordDim=2` in ESMF_GridGetCoord() references the latitude dimension.)

54.12 ESMF_CUBEDSPHERECALC

DESCRIPTION:
Indicates the method used to calculate coordinates during cubed sphere creation.

The type of this flag is:

`type(ESMF_CubedSphereCalc_Flag)`

The valid values are:

ESMF_CUBEDSPHERECALC_1TILE: This is the original method used to calculate coordinates for the ESMF cubed sphere. It uses an array the size of one tile of the cubed sphere on each PET to calculate coordinates and ensure their symmetry.

ESMF_CUBEDSPHERECALC_LOCAL: This method just uses an array large enough to hold the local cubed sphere coordinates on each PET. It relies on careful design of loops and calculation to ensure symmetry of the coordinates. This method will in general use less memory than ESMF_CUBEDSPHERECALC_1TILE.

54.13 ESMF_DATACOPY

DESCRIPTION:
Indicates how data references, copies, and memory allocations to hold data are handled.

The type of this flag is:

`type(ESMF_DataCopy_Flag)`

The valid values are:

ESMF_DATACOPY_ALLOC Create a new allocation that is sufficient in size to hold the data. However, no data is copied and the allocation is returned uninitialized.

ESMF_DATACOPY_REFERENCE Reference the data within the existing allocation.

ESMF_DATACOPY_VALUE Copy the data to another allocation. If needed create the new allocation.

54.14 ESMF_DECOMP

DESCRIPTION:
Indicates how DistGrid elements are decomposed over DEs.

The type of this flag is:

`type(ESMF_Decomp_Flag)`

The valid values are:

ESMF_DECOMP_BALANCED Decompose elements as balanced as possible across DEs. The maximum difference in number of elements per DE is 1, with the extra elements on the lower DEs.

ESMF_DECOMP_CYCLIC Decompose elements cyclically across DEs.

ESMF_DECOMP_RESTFIRST Divide elements over DEs. Assign the rest of this division to the first DE.

ESMF_DECOMP_RESTLAST Divide elements over DEs. Assign the rest of this division to the last DE.

ESMF_DECOMP_SYMMEDGEMAX Decompose elements across the DEs in a symmetric fashion. Start with the maximum number of elements at the two edge DEs, and assign a decending number of elements to the DEs as the center of the decomposition is approached from both sides.

54.15 ESMF_DIRECTION

This flag is documented in section 45.2.1.

54.16 ESMF_DISTGRIDMATCH

This flag is documented in section 36.2.1.

54.17 ESMF_END

This flag is documented in section 16.2.1.

54.18 ESMF_EXTRAPMETHOD

DESCRIPTION:
Specify which extrapolation method to use on unmapped destination points after regridding.

The type of this flag is:

type(ESMF_ExtrapMethod_Flag)

The valid values are:

ESMF_EXTRAPMETHOD_NONE Indicates that no extrapolation should be done.

ESMF_EXTRAPMETHOD_NEAREST_IDAVG Inverse distance weighted average. Here the value of an unmapped destination point is the weighted average of the closest N source points. The weight is the reciprocal of the distance of the source point from the destination point raised to a power P. All the weights contributing to one destination point are normalized so that they sum to 1.0. The user can choose N and P when using this method, but defaults are also provided. This extrapolation method is not supported with conservative regrid methods (e.g. ESMF_REGRIDMETHOD_CONSERVE).

ESMF_EXTRAPMETHOD_NEAREST_STOD Nearest source to destination. Here the value of each unmapped destination point is set to the value of the closest source point. This extrapolation method is not supported with conservative regrid methods (e.g. ESMF_REGRIDMETHOD_CONSERVE).

ESMF_EXTRAPMETHOD_NEAREST_D Nearest mapped destination to unmapped destination. Here the value of each unmapped destination point is set to the value of the closest mapped (i.e. regridded) destination point. This extrapolation method is not supported with conservative regrid methods (e.g. ESMF_REGRIDMETHOD_CONSERVE).

ESMF_EXTRAPMETHOD_CREEP Creep fill. Here unmapped destination points are filled by moving data from mapped locations to neighboring unmapped locations. The data filled into a new location is the average of its already filled neighbors' values. This process is repeated for a user specified number of levels (e.g. in ESMF_FieldRegridStore() this is specified via the extrapNumLevels argument). This extrapolation method is not supported with conservative regrid methods (e.g. ESMF_REGRIDMETHOD_CONSERVE).

ESMF_EXTRAPMETHOD_CREEP_NRST_D Creep fill with nearest destination. Here unmapped destination points are filled using creep fill as described under that entry (see ESMF_EXTRAPMETHOD_CREEP above), any points not filled by creep fill are then set to the closest filled or mapped destination point as if the nearest destination method (see ESMF_EXTRAPMETHOD_NEAREST_D above) were run after the initial regridding followed by creep fill. Like creep fill, this method is repeated for a user specified number of levels (e.g. in ESMF_FieldRegridStore() this is specified via the extrapNumLevels argument). This extrapolation method is not supported with conservative regrid methods (e.g. ESMF_REGRIDMETHOD_CONSERVE).

54.19 ESMF_FIELDSTATUS

This flag is documented in section 26.2.1.

54.20 ESMF_FILEFORMAT

DESCRIPTION:
This flag is used in ESMF_GridCreate and ESMF_MeshCreate functions. It is also used in the ESMF_RegridWeightGen API as an optional argument.

The type of this flag is:

type(ESMF_FileFormat_Flag)

The valid values are:

ESMF_FILEFORMAT_CFGRID A single tile logically rectangular grid file that follows the NetCDF CF convention. See section 12.8.3 for more detailed description.

ESMF_FILEFORMAT_ESMFMESH ESMF unstructured grid file format. This format was developed by the ESMF team to match the capabilities of the Mesh class and to be efficient to convert to that class. See section 12.8.2 for more detailed description.

ESMF_FILEFORMAT_GRIDSPEC Equivalent to `ESMF_FILEFORMAT_CFGRID` flag.

ESMF_FILEFORMAT_MOSAIC This format is a proposed extension to the CF-conventions for grid mosaics consisting of multiple logically rectangular grid tiles. See section 12.8.5 for more detailed description.

ESMF_FILEFORMAT_SCRIP SCRIP format grid file. The SCRIP format is the format accepted by the SCRIP regridding tool [7]. See section12.8.1 for more detailed description.

ESMF_FILEFORMAT_UGRID CF-convention unstructured grid file format. This format is a proposed extension to the CF-conventions for unstructured grid data model. See section 12.8.4 for more detailed description.

54.21 ESMF_FILEMODE

DESCRIPTION:
This flag is used to indicate which mode to use when writing a weight file.

The type of this flag is:

`type(ESMF_FileMode_Flag)`

The valid values are:

ESMF_FILEMODE_BASIC Indicates that only the factorList and factorIndexList should be written.

ESMF_FILEMODE_WITHAUX Indicates that grid center coordinates should also be written.

54.22 ESMF_FILESTATUS

DESCRIPTION:
This flag is used in ESMF I/O functions. It's use is similar to the `status` keyword in the Fortran `open` statement.

The type of this flag is:

`type(ESMF_FileStatus_Flag)`

The valid values are:

ESMF_FILESTATUS_NEW The file must not exist, it will be created.

ESMF_FILESTATUS_OLD The file must exist.

ESMF_FILESTATUS_REPLACE If the file exists, all of its contents will be deleted before writing. If the file does not exist, it will be created.

ESMF_FILESTATUS_UNKNOWN The value is treated as if it were `ESMF_FILESTATUS_OLD` if the corresponding file already exists. Otherwise, the value is treated as if it were `ESMF_FILESTATUS_NEW`.

54.23 ESMF_GEOMTYPE

DESCRIPTION:
Different types of geometries upon which an ESMF Field or ESMF Fieldbundle may be built.

The type of this flag is:

`type(ESMF_GeomType_Flag)`

The valid values are:

ESMF_GEOMTYPE_GRID An ESMF_Grid, a structured grid composed of one or more logically rectangular tiles.

ESMF_GEOMTYPE_MESH An ESMF_Mesh, an unstructured grid.

ESMF_GEOMTYPE_XGRID An ESMF_XGrid, an exchange grid.

ESMF_GEOMTYPE_LOCSTREAM An ESMF_LocStream, a disconnected series of points with associated key values.

54.24 ESMF_GRIDCONN

This flag is documented in section 31.2.1.

54.25 ESMF_GRIDITEM

This flag is documented in section 31.2.2.

54.26 ESMF_GRIDMATCH

This flag is documented in section 31.2.3.

54.27 ESMF_GRIDSTATUS

This flag is documented in section 31.2.4.

54.28 ESMF_HCONFIGMATCH

This flag is documented in section 48.2.1.

54.29 ESMF_INDEX

DESCRIPTION:
Indicates whether index is local (per DE) or global (per object).

The type of this flag is:

`type(ESMF_Index_Flag)`

The valid values are:

ESMF_INDEX_DELOCAL Indicates that DE-local index space starts at lower bound 1 for each DE.

ESMF_INDEX_GLOBAL Indicates that global indices are used. This means that DE-local index space starts at the global lower bound for each DE.

ESMF_INDEX_USER Indicates that the DE-local index bounds are explicitly set by the user.

54.30 ESMF_IOFMT

DESCRIPTION:
Indicates I/O format options that are currently supported.

The type of this flag is:

```
type(ESMF_IOFmt_Flag)
```

The valid values are:

ESMF_IOFMT_NETCDF NETCDF and PNETCDF (cdf1) format in NETCDF-3 "classic" format. Use of PNETCDF is prioritized if available.

ESMF_IOFMT_NETCDF_64BIT_OFFSET NETCDF and PNETCDF (cdf2) format. This format allows files greater than 4GiB in NETCDF-3 "classic" format. Use of PNETCDF is prioritized if available.

ESMF_IOFMT_NETCDF_64BIT_DATA NETCDF and PNETCDF (cdf5) format. This allows individual records greater than 4GiB in NETCDF-3 "classic" format. Use of PNETCDF is prioritized if available.

ESMF_IOFMT_NETCDF4P NETCDF-4 (HDF-5) format. If a NETCDF parallel library is available, writes will be in parallel. Writes will use one I/O PET per SSI (node) of the ESMF VM that calls the I/O operation. Otherwise writes will be serial.

ESMF_IOFMT_NETCDF4C NETCDF-4 (HDF-5) format with lossless compression from HDF-5 applied. This is only available as a serial option, even if a parallel NETCDF library is available.

54.31 ESMF_IO_NETCDF_PRESENT

DESCRIPTION:
Indicates whether netcdf feature support has been enabled within the current ESMF build.

The type of this flag is:

```
logical
```

The valid values are:

.true. Netcdf features are enabled.

.false. Netcdf features are not enabled.

54.32 ESMF_IO_PIO_PRESENT

DESCRIPTION:
Indicates whether PIO (parallel I/O) feature support has been enabled within the current ESMF build.

The type of this flag is:

```
logical
```

The valid values are:

.true. PIO features are enabled.

.false. PIO features are not enabled.

54.33 ESMF_IO_PNETCDF_PRESENT

DESCRIPTION:
Indicates whether parallel netcdf feature support has been enabled within the current ESMF build.

The type of this flag is:

```
logical
```

The valid values are:

.true. Parallel NETCDF features are enabled.

.false. Parallel NETCDF features are not enabled.

54.34 ESMF_ITEMORDER

DESCRIPTION:
Specifies the order of items in a list.

The type of this flag is:

```
type(ESMF_ItemOrder_Flag)
```

The valid values are:

ESMF_ITEMORDER_ABC The items are in alphabetical order, according to their names.

ESMF_ITEMORDER_ADDORDER The items are in the order in which they were added to the container.

54.35 ESMF_KIND

DESCRIPTION:
Named constants to be used as *kind-parameter* in Fortran variable declarations. For example:

```
  integer(ESMF_KIND_I4)       :: integerVariable
  integer(kind=ESMF_KIND_I4)  :: integerVariable
```

```
real(ESMF_KIND_R4)          :: realVariable
real(kind=ESMF_KIND_R4)     :: realVariable
```

The Fortran standard does not mandate what numeric values correspond to actual number of bytes allocated for the various kinds. The following constants are defined by ESMF to be correct across the supported Fortran compilers. Note that not all compilers support every kind listed below; in particular 1 and 2 byte integers can be problematic.

The type of these named constants is:

`integer`

The named constants are:

ESMF_KIND_I1 Kind-parameter for 1 byte integer.

ESMF_KIND_I2 Kind-parameter for 2 byte integer.

ESMF_KIND_I4 Kind-parameter for 4 byte integer.

ESMF_KIND_I8 Kind-parameter for 8 byte integer.

ESMF_KIND_R4 Kind-parameter for 4 byte real.

ESMF_KIND_R8 Kind-parameter for 8 byte real.

54.36 ESMF_LINETYPE

DESCRIPTION:
This argument allows the user to select the path of the line which connects two points on the surface of a sphere. This in turn controls the path along which distances are calculated and the shape of the edges that make up a cell.

The type of this flag is:

`type(ESMF_LineType_Flag)`

The valid values are:

ESMF_LINETYPE_CART Cartesian line. When this option is specified distances are calculated in a straight line through the 3D Cartesian space in which the sphere is embedded. Cells are approximated by 3D planes bounded by 3D Cartesian lines between their corner vertices. When calculating regrid weights, this line type is currently the default for non-conservative regrid methods (e.g. ESMF_REGRIDMETHOD_BILINEAR, ESMF_REGRIDMETHOD_PATCH, ESMF_REGRIDMETHOD_NEAREST_STOD, etc.).

ESMF_LINETYPE_GREAT_CIRCLE Great circle line. When this option is specified distances are calculated along a great circle path (the shortest distance between two points on a sphere surface). Cells are bounded by great circle paths between their corner vertices. When calculating regrid weights, this line type is currently the default for conservative regrid methods (e.g. ESMF_REGRIDMETHOD_CONSERVE, etc.).

54.37 ESMF_LOGERR

This flag is documented in section 49.2.1.

54.38 ESMF_LOGKIND

This flag is documented in section 49.2.2.

54.39 ESMF_LOGMSG

This flag is documented in section 49.2.3.

54.40 ESMF_MESHELEMTYPE

This flag is documented in section 33.2.1.

54.41 ESMF_MESHLOC

DESCRIPTION:
Used to indicate a specific part of a Mesh. This is commonly used to specify the part of the Mesh to create a Field on.

The type of this flag is:

`type(ESMF_MeshLoc)`

The valid values are:

ESMF_MESHLOC_NODE The nodes (also known as corners or vertices) of a Mesh.

ESMF_MESHLOC_ELEMENT The elements (also known as cells) of a Mesh.

54.42 ESMF_MESHOP

DESCRIPTION:
Specifies the spatial operation with two source Meshes, treating the Meshes as point sets.

The type of this flag is:

`type(ESMF_MeshOp_Flag)`

The valid values are:

ESMF_MESHOP_DIFFERENCE Calculate the difference of the two point sets from the source Meshes.

54.43 ESMF_MESHSTATUS

DESCRIPTION:
The ESMF Mesh class can exist in several states. The ESMF_MESHSTATUS flag is used to indicate which of these states a Mesh is currently in.

The type of this flag is:

`type(ESMF_MeshStatus_Flag)`

The valid values are:

ESMF_MESHSTATUS_UNINIT: The Mesh status is uninitialized. This might happen if the Mesh hasn't been created yet, or if the Mesh has been destroyed.

ESMF_MESHSTATUS_EMPTY: Status after a Mesh has been created with `ESMF_MeshEmptyCreate`. Only distribution information has been set in the Mesh. This object can be used in the `ESMF_MeshGet()` method to retrieve distgrids and the distgrid's presence.

ESMF_MESHSTATUS_STRUCTCREATED: Status after a Mesh has been through the first step of the three step creation process. The Mesh is now ready to have nodes added to it.

ESMF_MESHSTATUS_NODESADDED: Status after a Mesh has been through the second step of the three step creation process. The Mesh is now ready to be completed with elements.

ESMF_MESHSTATUS_COMPLETE: The Mesh has been completely created and can be used to create a Field. Further, if the internal Mesh memory hasn't been freed, then the Mesh should be usable in any Mesh functionality (e.g. regridding). The status of the internal Mesh memory can be checked using the `isMemFreed` argument to `ESMF_MeshGet()`.

54.44 ESMF_METHOD

DESCRIPTION:
Specify standard ESMF Component method.

The type of this flag is:

`type(ESMF_Method_Flag)`

The valid values are:

ESMF_METHOD_FINALIZE Finalize method.

ESMF_METHOD_INITIALIZE Initialize method.

ESMF_METHOD_READRESTART ReadRestart method.

ESMF_METHOD_RUN Run method.

ESMF_METHOD_WRITERESTART WriteRestart method.

54.45 ESMF_NORMTYPE

DESCRIPTION:
When doing conservative regridding (e.g. `ESMF_REGRIDMETHOD_CONSERVE`), this option allows the user to select the type of normalization used when producing the weights.

`type(ESMF_NormType_Flag)`

The valid values are:

ESMF_NORMTYPE_DSTAREA Destination area normalization. Here the weights are calculated by dividing the area of overlap of the source and destination cells by the area of the entire destination cell. In other words, the weight is the fraction of the entire destination cell which overlaps with the given source cell.

ESMF_NORMTYPE_FRACAREA Fraction area normalization. Here in addition to the weight calculation done for destination area normalization (`ESMF_NORMTYPE_DSTAREA`) the weights are also divided by the fraction that the destination cell overlaps with the entire source grid. In other words, the weight is the fraction of just the part of the destination cell that overlaps with the entire source mesh.

54.46 ESMF_PIN

This flag is documented in section 50.2.1.

54.47 ESMF_POLEKIND

This flag is documented in section 31.2.5.

54.48 ESMF_POLEMETHOD

DESCRIPTION:

When interpolating between two Grids which have been mapped to a sphere these can be used to specify the type of artificial pole to create on the source Grid during interpolation. Creating the pole allows destination points above the top row or below the bottom row of the source Grid to still be mapped.

The type of this flag is:

`type(ESMF_PoleMethod_Flag)`

The valid values are:

ESMF_POLEMETHOD_NONE No pole. Destination points which lie above the top or below the bottom row of the source Grid won't be mapped.

ESMF_POLEMETHOD_ALLAVG Construct an artificial pole placed in the center of the top (or bottom) row of nodes, but projected onto the sphere formed by the rest of the grid. The value at this pole is the average of all the source values surrounding the pole.

ESMF_POLEMETHOD_NPNTAVG Construct an artificial pole placed in the center of the top (or bottom) row of nodes, but projected onto the sphere formed by the rest of the grid. The value at this pole is the average of the N source nodes next to the pole and surrounding the destination point (i.e. the value may differ for each destination point). Here N is set by using the `regridPoleNPnts` parameter and ranges from 1 to the number of nodes around the pole. This option is useful for interpolating values which may be zeroed out by averaging around the entire pole (e.g. vector components).

ESMF_POLEMETHOD_TEETH No new pole point is constructed, instead the holes at the poles are filled by constructing triangles across the top and bottom row of the source Grid. This can be useful because no averaging occurs, however, because the top and bottom of the sphere are now flat, for a big enough mismatch between the size of the destination and source pole holes, some destination points may still not be able to be mapped to the source Grid.

54.49 ESMF_REDUCE

DESCRIPTION:
Indicates reduce operation.

The type of this flag is:

```
type(ESMF_Reduce_Flag)
```

The valid values are:

ESMF_REDUCE_SUM Use arithmetic sum to add all data elements.

ESMF_REDUCE_MIN Determine the minimum of all data elements.

ESMF_REDUCE_MAX Determine the maximum of all data elements.

54.50 ESMF_REGION

DESCRIPTION:
Specifies various regions in the data layout of an Array or Field object.

The type of this flag is:

```
type(ESMF_Region_Flag)
```

The valid values are:

ESMF_REGION_TOTAL Total allocated memory.

ESMF_REGION_SELECT Region of operation-specific elements.

ESMF_REGION_EMPTY The empty region contains no elements.

54.51 ESMF_REGRIDMETHOD

DESCRIPTION:
Specify which interpolation method to use during regridding. For a more detailed discussion of these methods, as well as ESMF regridding in general, see Section 24.2.

The type of this flag is:

```
type(ESMF_RegridMethod_Flag)
```

The valid values are:

ESMF_REGRIDMETHOD_BILINEAR Bilinear interpolation. Destination value is a linear combination of the source values in the cell which contains the destination point. The weights for the linear combination are based on the distance of destination point from each source value.

ESMF_REGRIDMETHOD_PATCH Higher-order patch recovery interpolation. Destination value is a weighted average of 2D polynomial patches constructed from cells surrounding the source cell which contains the destination point. This method typically results in better approximations to values and derivatives than bilinear. However, because of its larger stencil, it also results in a much larger interpolation matrix (and thus routeHandle) than the bilinear.

ESMF_REGRIDMETHOD_NEAREST_STOD In this version of nearest neighbor interpolation each destination point is mapped to the closest source point. A given source point may go to multiple destination points, but no destination point will receive input from more than one source point.

ESMF_REGRIDMETHOD_NEAREST_DTOS In this version of nearest neighbor interpolation each source point is mapped to the closest destination point. A given destination point may receive input from multiple source points, but no source point will go to more than one destination point.

ESMF_REGRIDMETHOD_CONSERVE First-order conservative interpolation. The main purpose of this method is to preserve the integral of the field between the source and destination. Will typically give a less accurate approximation to the individual field values than the bilinear or patch methods. The value of a destination cell is calculated as the weighted sum of the values of the source cells that it overlaps. The weights are determined by the amount the source cell overlaps the destination cell. Needs corner coordinate values to be provided in the Grid. Currently only works for Fields created on the Grid center stagger or the Mesh element location.

ESMF_REGRIDMETHOD_CONSERVE_2ND Second-order conservative interpolation. As with first-order, preserves the integral of the value between the source and destination. However, typically produces a smoother more accurate result than first-order. Also like first-order, the value of a destination cell is calculated as the weighted sum of the values of the source cells that it overlaps. However, second-order also includes additional terms to take into account the gradient of the field across the source cell. Needs corner coordinate values to be provided in the Grid. Currently only works for Fields created on the Grid center stagger or the Mesh element location.

54.52 ESMF_REGRIDSTATUS

DESCRIPTION:

These values can be output during regridding (e.g. from `ESMF_FieldRegridStore()` via the `dstStatusField` argument). They indicate the status of each destination location.

The type of this flag is:

```
integer(ESMF_KIND_I4)
```

The valid values for all regrid methods are:

ESMF_REGRIDSTATUS_DSTMASKED The destination location is masked, so no regridding has been peformed on it. This status has a numeric value of 0.

ESMF_REGRIDSTATUS_SRCMASKED The destination location is within a masked part of the source grid, so no regridding has been performed on it. This status has a numeric value of 1.

ESMF_REGRIDSTATUS_OUTSIDE The destination location is outside the source grid, so no regridding has been performed on it. This status has a numeric value of 2.

ESMF_REGRIDSTATUS_MAPPED The destination location is within the unmasked source grid, and so has been regridded (i.e. there is an entry for it within the factorIndexList or routeHandle). This status has a numeric value of 4.

ESMF_REGRIDSTATUS_EXMAPPED The destination location was not within the unmasked source grid, and so typically it wouldn't be regridded. However, extrapolation was used, and so it has been regridded (i.e. there is an entry for it within the factorIndexList or routeHandle). This status has a numeric value of 8.

In addition to the above, regridding using the conservative method can result in other values. The reason for this is that in that method one destination cell can overlap multiple source cells, so a single destination can have a combination of values. The following are the additional values that apply to the conservative method:

ESMF_REGRIDSTATUS_SMSK_OUT The destination cell overlaps a masked source cell, and extends outside the source grid. This status has a numeric value of 3.

ESMF_REGRIDSTATUS_SMSK_MP The destination cell overlaps a masked source cell, and an unmasked source cell. (Because it overlaps with the unmasked source grid, there will be an entry for the destination cell within the factorIndexList or routeHandle). This status has a numeric value of 5.

ESMF_REGRIDSTATUS_OUT_MP The destination cell overlaps an unmasked source cell, and extends outside the source grid. (Because it overlaps with the unmasked source grid, there will be an entry for the destination cell within the factorIndexList or routeHandle). This status has a numeric value of 6.

ESMF_REGRIDSTATUS_SMSK_OUT_MP The destination cell overlaps a masked source cell, an unmasked source cell, and extends outside the source grid. (Because it overlaps with the unmasked source grid, there will be an entry for the destination cell within the factorIndexList or routeHandle). This status has a numeric value of 7.

54.53 ESMF_ROUTESYNC

DESCRIPTION:

Switch between blocking and non-blocking execution of RouteHandle based communication calls. Every RouteHandle based communication method contains an optional argument `routesyncflag` that is of type `ESMF_RouteSync_Flag`.

The type of this flag is:

`type(ESMF_RouteSync_Flag)`

The valid values are:

ESMF_ROUTESYNC_BLOCKING Execute a precomputed communication pattern in blocking mode. This mode guarantees that when the method returns all PET-local data transfers, both in-bound and out-bound, have finished.

ESMF_ROUTESYNC_NBSTART Start executing a precomputed communication pattern in non-blocking mode. When a method returns from being called in this mode, it guarantees that all PET-local out-bound data has been transferred. It is now safe for the user to overwrite out-bound data elements. No guarantees are made for in-bound data elements at this stage. It is unsafe to access these elements until a call in `ESMF_ROUTESYNC_NBTESTFINISH` mode has been issued and has returned with `finishedflag` equal to .true., or a call in `ESMF_ROUTESYNC_NBWAITFINISH` mode has been issued and has returned.

ESMF_ROUTESYNC_NBTESTFINISH Test whether the transfer of data of a precomputed communication pattern, started with `ESMF_ROUTESYNC_NBSTART`, has completed. Finish up as much as possible and set the `finishedflag` to `.true.` if *all* data operations have completed, or `.false.` if there are still outstanding transfers. Only after a `finishedflag` equal to `.true.` has been returned is it safe to access any of the in-bound data elements.

ESMF_ROUTESYNC_NBWAITFINISH Wait (i.e. block) until the transfer of data of a precomputed communication pattern, started with `ESMF_ROUTESYNC_NBSTART`, has completed. Finish up *all* data operations and set the returned `finishedflag` to `.true.`. It is safe to access any of the in-bound data elements once the call has returned.

ESMF_ROUTESYNC_CANCEL Cancel outstanding transfers for a precomputed communication pattern.

54.54 ESMF_SERVICEREPLY

This flag is documented in section 50.2.2.

54.55 ESMF_STAGGERLOC

This flag is documented in section 31.2.6.

54.56 ESMF_STARTREGION

DESCRIPTION:
Specifies the start of the effective halo region of an Array or Field object.

The type of this flag is:

`type(ESMF_StartRegion_Flag)`

The valid values are:

ESMF_STARTREGION_EXCLUSIVE Region of elements that are exclusively owned by the local DE.

ESMF_STARTREGION_COMPUTATIONAL User defined region, greater or equal to the exclusive region.

54.57 ESMF_STATEINTENT

This flag is documented in section 21.2.1.

54.58 ESMF_STATEITEM

This flag is documented in section 21.2.2.

54.59 ESMF_SYNC

DESCRIPTION:
Indicates method blocking behavior and PET synchronization for VM communication methods, as well as for standard Component methods, such as Initialize(), Run() and Finalize().

For VM communication calls the `ESMF_SYNC_BLOCKING` and `ESMF_SYNC_NONBLOCKING` modes provide behavior that is practically identical to the blocking and non-blocking communication calls familiar from MPI.

The details of how the blocking mode setting affects Component methods are more complex. This is a consequence of the fact that ESMF Components can be executed in threaded or non-threaded mode. However, in the default, non-threaded case, where an ESMF application runs as a pure MPI or mpiuni program, most of the complexity is removed.

See the **VM** item in 6.6 for an explanation of the PET and VAS concepts used in the following descriptions.

The type of this flag is:

`type(ESMF_Sync_Flag)`

The valid values are:

ESMF_SYNC_BLOCKING *Communication calls:* The called method will block until all (PET-)local operations are complete. After the return of a blocking communication method it is safe to modify or use all participating local data.

Component calls: The called method will block until all PETs of the VM have completed the operation.

For a non-threaded, pure MPI component the behavior is identical to calling a barrier before returning from the method. Generally this kind of rigid synchronization is not the desirable mode of operation for an MPI application, but may be useful for application debugging. In the opposite case, where all PETs of the component are running as threads in shared memory, i.e. in a single VAS, strict synchronization of all PETs is required to prevent race conditions.

ESMF_SYNC_VASBLOCKING *Communication calls:* Not available for communication calls.

Component calls: The called method will block each PET until all operations in the PET-local VAS have completed.

This mode is a combination of `ESMF_SYNC_BLOCKING` and `ESMF_SYNC_NONBLOCKING` modes. It provides a default setting that leads to the typically desirable behavior for pure MPI components as well as those that share address spaces between PETs.

For a non-threaded, pure MPI component each PET returns independent of the other PETs. This is generally the expected behavior in the pure MPI case where calling into a component method is practically identical to a subroutine call without extra synchronization between the processes.

In the case where some PETs of the component are running as threads in shared memory `ESMF_SYNC_VASBLOCKING` becomes identical to `ESMF_SYNC_BLOCKING` within thread groups, to prevent race conditions, while there is no synchronization between the thread groups.

ESMF_SYNC_NONBLOCKING *Communication calls:* The called method will not block but returns immediately after initiating the requested operation. It is unsafe to modify or use participating local data before all local operations have completed. Use the `ESMF_VMCommWait()` or `ESMF_VMCommQueueWait()` method to block the local PET until local data access is safe again.

Component calls: The behavior of this mode is fundamentally different for threaded and non-threaded components, independent on whether the components use shared memory or not. The `ESMF_SYNC_NONBLOCKING` mode is the most complex mode for calling component methods and should only be used if the extra control, described below, is absolutely necessary.

For non-threaded components (the ESMF default) calling a component method with `ESMF_SYNC_NONBLOCKING` is identical to calling it with `ESMF_SYNC_VASBLOCKING`. However, different than for `ESMF_SYNC_VASBLOCKING`, a call to `ESMF_GridCompWait()` or `ESMF_CplCompWait()` is required in order to deallocate memory internally allocated for the `ESMF_SYNC_NONBLOCKING` mode.

For threaded components the calling PETs of the parent component will not be blocked and return immediately after initiating the requested child component method. In this scenario parent and child components will run concurrently in identical VASs. This is the most complex mode of operation. It is unsafe to modify or use VAS local data that may be accessed by concurrently running components until the child component method has completed. Use the appropriate `ESMF_GridCompWait()` or `ESMF_CplCompWait()` method to block the local parent PET until the child component method has completed in the local VAS.

54.60 ESMF_TERMORDER

DESCRIPTION:
Specifies the order of source terms in a destination sum, e.g. during sparse matrix multiplication.

The type of this flag is:

`type(ESMF_TermOrder_Flag)`

The valid values are:

ESMF_TERMORDER_SRCSEQ The source terms are in strict ascending order according to their source sequence index.

ESMF_TERMORDER_SRCPET The source terms are first ordered according to their distribution across the source side PETs: for each destination PET the source PET order starts with the localPet and decrements from there, modulo petCount, until all petCount PETs are accounted for. The term order within each source PET is given by the source term sequence index.

ESMF_TERMORDER_FREE There is no prescribed term order. The source terms may be summed in any order that optimizes performance.

54.61 ESMF_TYPEKIND

DESCRIPTION:
Named constants used to indicate type and kind combinations supported by the overloaded ESMF interfaces. The corresponding Fortran kind-parameter constants are described in section 54.35.

The type of these named constants is:

`type(ESMF_TypeKind_Flag)`

The named constants numerical types are:

ESMF_TYPEKIND_I1 Indicates 1 byte integer.
(Only available if ESMF was built with `ESMF_NO_INTEGER_1_BYTE = FALSE`. This is *not* the default.)

ESMF_TYPEKIND_I2 Indicates 2 byte integer.
(Only available if ESMF was built with `ESMF_NO_INTEGER_2_BYTE = FALSE`. This is *not* the default.)

ESMF_TYPEKIND_I4 Indicates 4 byte integer.

ESMF_TYPEKIND_I8 Indicates 8 byte integer.

ESMF_TYPEKIND_R4 Indicates 4 byte real.

ESMF_TYPEKIND_R8 Indicates 8 byte real.

The named constants non-numerical types are:

ESMF_TYPEKIND_LOGICAL Indicates a logical value.

ESMF_TYPEKIND_CHARACTER Indicates a character string.

54.62 ESMF_UNMAPPEDACTION

DESCRIPTION:
Indicates what action to take with respect to unmapped destination points and the entries of the sparse matrix that correspond to these points.

The type of this flag is:

`type(ESMF_UnmappedAction_Flag)`

The valid values are:

ESMF_UNMAPPEDACTION_ERROR An error is issued when there exist destination points in a regridding operation that are not mapped by corresponding source points.

ESMF_UNMAPPEDACTION_IGNORE Destination points which do not have corresponding source points are ignored and zeros are used for the entries of the sparse matrix that is generated.

54.63 ESMF_VERSION

DESCRIPTION:
The following named constants define the precise version of ESMF in use.

ESMF_VERSION_BETASNAPSHOT Constant of type `logical` indicating beta snapshot phase (`.true.` for any version during the pre-release development phase, `.false.` for any released version of the software).

ESMF_VERSION_MAJOR Constant of type `integer` indicating the major version number (e.g. 5 for v5.2.0r).

ESMF_VERSION_MINOR Constant of type `integer` indicating the minor version number (e.g. 2 for v5.2.0r).

ESMF_VERSION_PATCHLEVEL Constant of type `integer` indicating the patch level of a specific revision (e.g. 0 for v5.2.0r, or 1 for v5.2.0rp1).

ESMF_VERSION_PUBLIC Constant of type `logical` indicating public vs. internal release status (e.g. `.true.` for v5.2.0r, or `.false.` for v5.2.0).

ESMF_VERSION_REVISION Constant of type `integer` indicating the revision number (e.g. 0 for v5.2.0r).

ESMF_VERSION_STRING Constant of type `character` holding the exact release version string (e.g. "5.2.0r").

54.64 ESMF_VMEPOCH

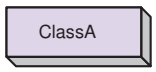
This flag is documented in section 51.2.1.

54.65 ESMF_XGRIDSIDE

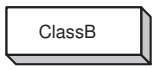
This flag is documented in section 34.2.1.

55 Appendix B: A Brief Introduction to UML

The schematic below shows the Unified Modeling Language (UML) notation for the class diagrams presented in this *Reference Manual*. For more on UML, see references such as *The Unified Modeling Language Reference Manual*, Rumbaugh et al, [19].



Public class. This is a class whose methods can be called by the user. In Fortran a public class is usually associated with a derived type and a corresponding module that contains class methods and flags.



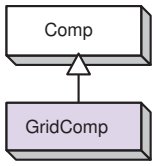
Private class. This type of class does not have methods that should be called by the user. Like a public class it is usually associated with a derived type and a corresponding module.

A line indicates some sort of association among classes.

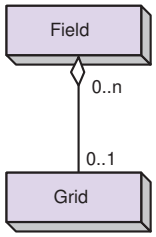
A hollow diamond at one end of a line drawn between classes represents an association called aggregation. Aggregation is a part-whole relationship that can be read as "the class at the end of the line without the diamond is part of the class at the end of the line with the diamond." The class that is the "part" can be created and destroyed separately, and it is usually implemented as a reference contained with the structure of the class that is the "whole."

A filled diamond at one end of a line drawn between classes represents an association called composition. Composition is a part-whole relationship that is similar to aggregation, but stronger. It implies that that class that is the "part" is created and destroyed by the class that is the "whole." It is often implemented as a structure within part of the contiguous memory of a larger structure.

Multiplicity indicators at association line ends show how many classes on the one end are associated with how many classes on the other end.



The triangle indicates an inheritance relationship. Inheritance means that a child class shares a set of characteristics (such as the same attributes or methods) with a parent class. The child can specialize and extend the behavior of the parent. This diagram shows a GridComp class that inherits from a more general Comp class.



This simple diagram shows that a public class called Field is associated with another public class, called Grid. The aggregation relationship indicated by the unfilled diamond means that a Field contains a Grid, but that a Grid can be created and destroyed outside of a Field. The diagram multiplicities show that a Field can be associated with no Grid or with one Grid, but that a single Grid can be associated with any number of Fields.

56 Appendix C: ESMF Error Return Codes

The tables below show the possible error return codes for Fortran and C methods.

57 Appendix D: Attribute Class Legacy API

Documentation for the legacy `ESMF_Attribute` Class. It is recommended that users migrate to the `ESMF_Info` class (see section 40).
Notice that a few aspects of the legacy Attribute API have been modified compared to its original implementation. These changes were necessary as a consequence of the changed backend to `ESMF_Info`:

- The `ESMF_AttributeSet()` method now supports setting attributes that were not previously added via `ESMF_AttributeAdd()`. In other words, the `ESMF_AttributeAdd()` method has become optional.
- There are overloads of the `ESMF_AttributeSet()` and `ESMF_AttributeGet()` with optional `convention` and `purpose` arguments. Both arguments must either be present or absent. Any other combination is handled as an error condition. For the case that both `convention` and `purpose` arguments are absent, and no Attribute Package is specified, the default JSON key prefix is "/ESMF/General".

57.1 Constants

57.1.1 ESMF_ATTCOPY

DEPRECATED CLASS!
The entire ESMF_Attribute class has been deprecated and is scheduled for removal with ESMF 9.0.0. This includes all of the class derived types, named constants, and methods. Please use the replacment class `ESMF_Info`, section `40` instead!

DESCRIPTION:
Indicates which type of copy behavior is used when copying ESMF Attribute objects.

The type of this flag is:

`type(ESMF_AttCopy_Flag)`

The valid values are:

ESMF_ATTCOPY_REFERENCE The destination Attribute hierarchy becomes a reference copy of the Attribute hierarchy of the source object. Any further changes to one will also be reflected in the other.

ESMF_ATTCOPY_VALUE All of the Attributes and Attribute packages of the source object will be copied by value to the destination object. None of the Attribute links to the Attribute hierarchies of other objects are copied to the destination object.

57.1.2 ESMF_ATTGETCOUNT

DEPRECATED CLASS!
The entire ESMF_Attribute class has been deprecated and is scheduled for removal with ESMF 9.0.0. This includes

all of the class derived types, named constants, and methods. Please use the replacment class `ESMF_Info`, section `40` instead!

DESCRIPTION:
Indicates which type of Attribute object count to return.

The type of this flag is:

`type(ESMF_AttGetCountFlag)`

The valid values are:

ESMF_ATTGETCOUNT_ATTRIBUTE This option will allow the routine to return the number of single Attributes.

ESMF_ATTGETCOUNT_ATTPACK This option will allow the routine to return the number of Attribute packages.

ESMF_ATTGETCOUNT_TOTAL This option will allow the routine to return the total number of Attributes.

57.1.3 ESMF_ATTWRITE

DEPRECATED CLASS!
The entire ESMF_Attribute class has been deprecated and is scheduled for removal with ESMF 9.0.0. This includes all of the class derived types, named constants, and methods. Please use the replacment class `ESMF_Info`, section `40` instead!

DESCRIPTION:
Indicates which file format to use in the write operation.

The type of this flag is:

`type(ESMF_AttWriteFlag)`

The valid values are:

ESMF_ATTWRITE_JSON This option will allow the routine to write in JSON format.

57.2 Class API