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

April 14, 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.

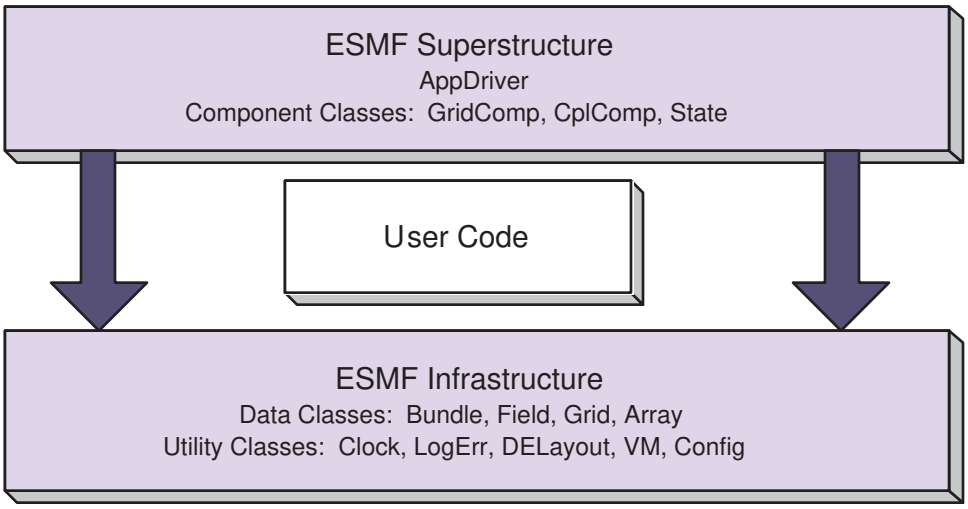
2 The ESMF Reference Manual for Fortran

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.* [**?**]

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.

6.3.1 ESMF_NamedAlias - Generate a Named Alias

INTERFACE:

```
function ESMF_NamedAlias(object, name, rc)
```

RETURN VALUE:

```
type(ESMF_*)        :: ESMF_NamedAlias
```

ARGUMENTS:

```
type(ESMF_*),       intent(in)            :: object
character(len = *), intent(in),  optional :: name
integer,            intent(out), optional :: rc
```

DESCRIPTION:

Generate a named alias to `object`. The supported classes are:

- `ESMF_State`
- `ESMF_GridComp`
- `ESMF_CplComp`
- `ESMF_SciComp`
- `ESMF_FieldBundle`
- `ESMF_Field`
- `ESMF_ArrayBundle`
- `ESMF_Array`

The arguments are:

object The incoming object for which a named alias is generated.

[name] The name of the named alias. By default use the name of `object`.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

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 **??**, 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 **??**, 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 **??**.

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 **??**). 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 [**?**], 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 **??**.

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 **??** 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 **??**.

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 [**?**] [**?**]. 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 [**?**].

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 **??**.

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 [**?**].

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 **??**.

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, N
                    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, N
                    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, N
                    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 [**?**] 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 **??**. 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`:

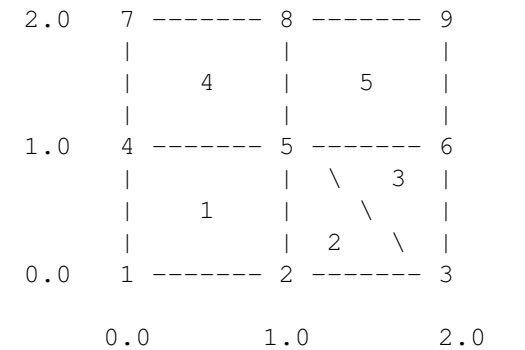
```
netcdf catchments_esmf1 {
dimensions:
        nodeCount = 1824345 ;
        elementCount = 68127 ;
```

```
        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 **??**. 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/z11/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 **??**). 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 **??** 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

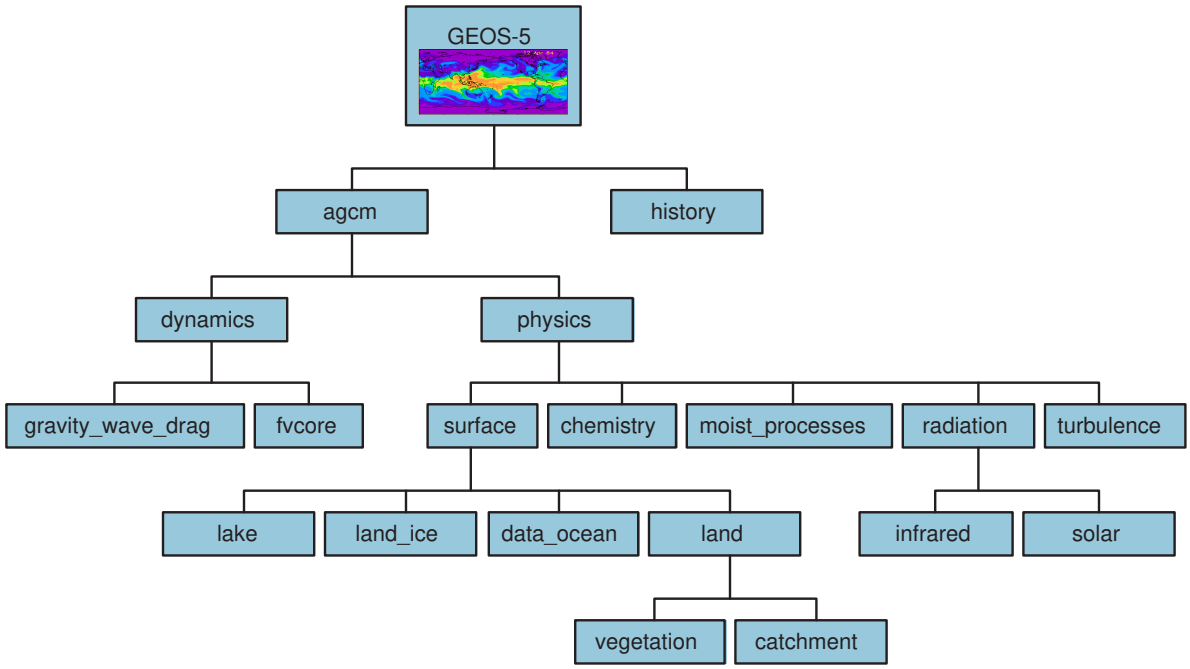
There are a small number of classes in the ESMF superstructure:

- **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.



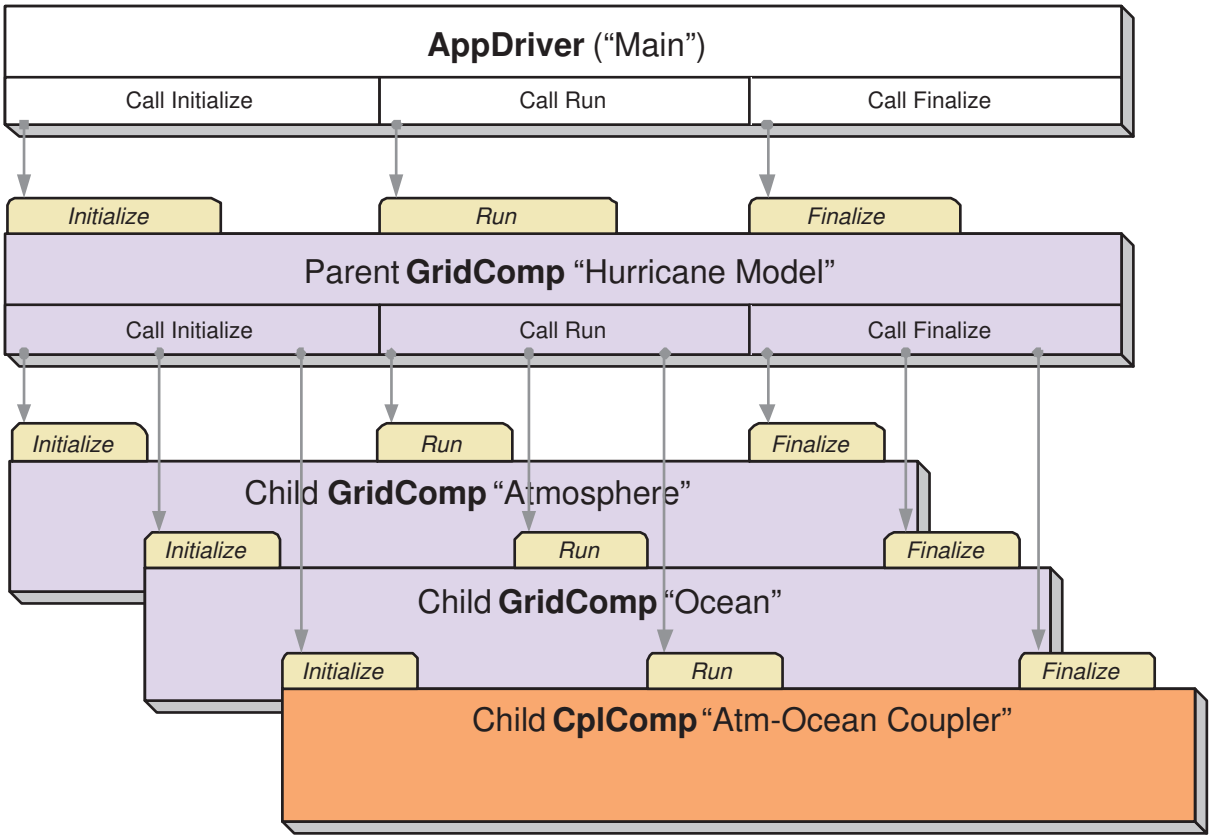
An ESMF coupled application typically involves a parent Gridded Component, two or more child Gridded Components and one or more Coupler Components.

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.

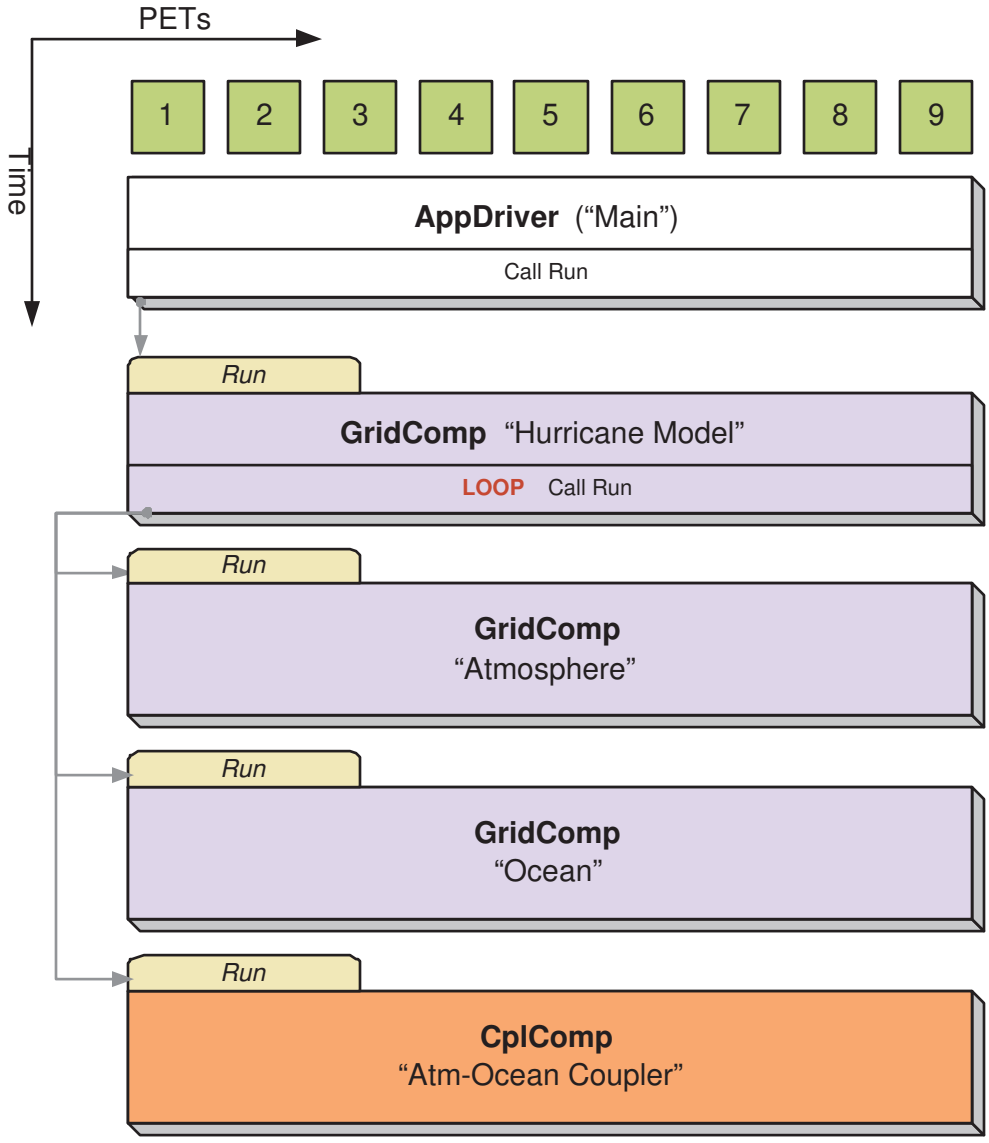


Figure 5: Schematic of the run method of a coupled application, with an "Atmosphere" and an "Ocean" Gridded Component running concurrently with an "Atm-Ocean Coupler." The top-level "Hurricane Model" Gridded Component contains the sequencing information and time advancement loop. The application driver, 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.

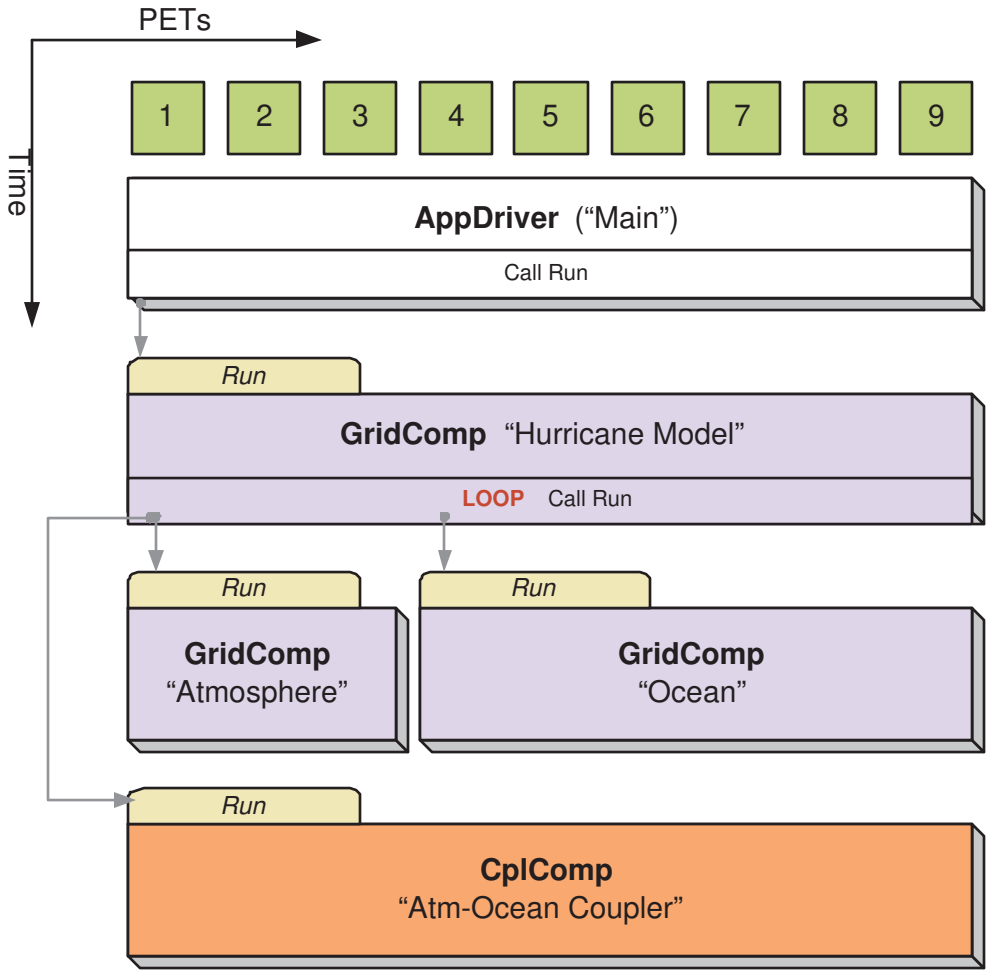
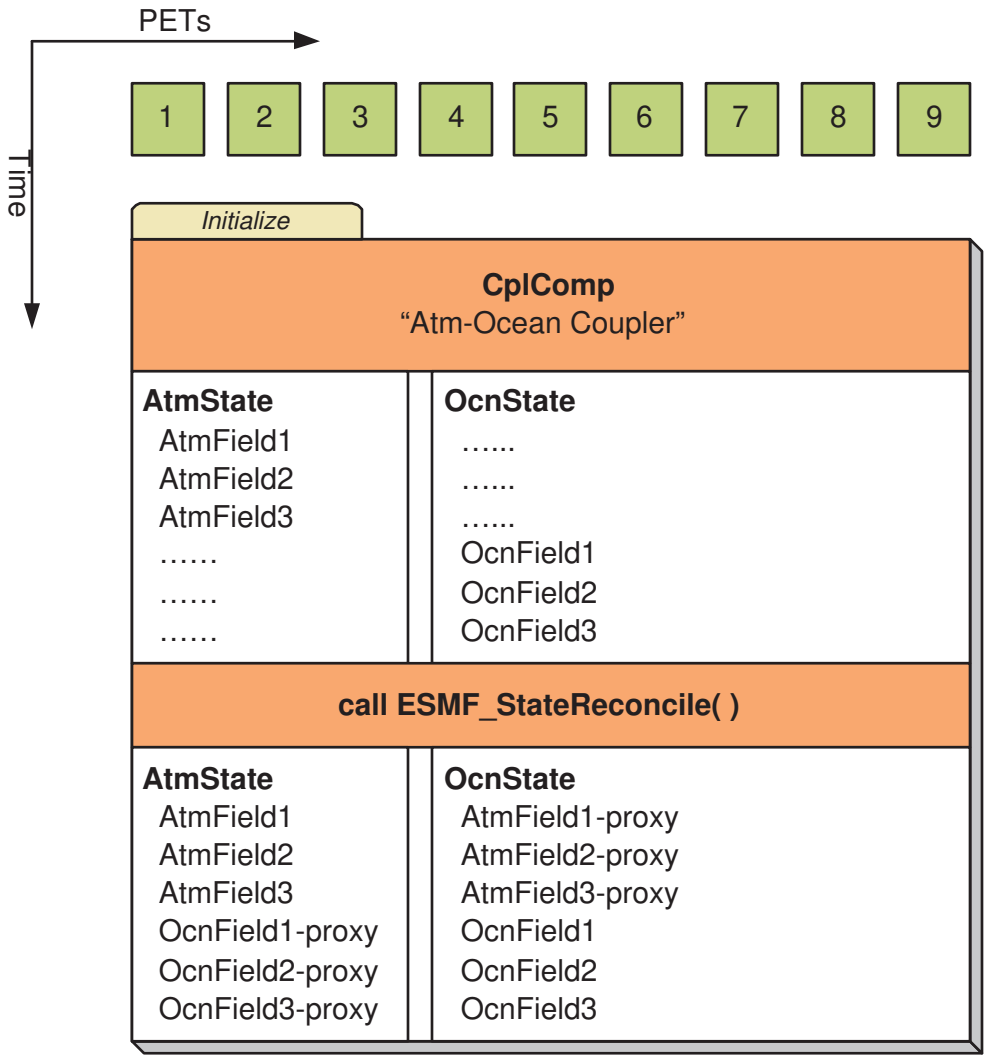
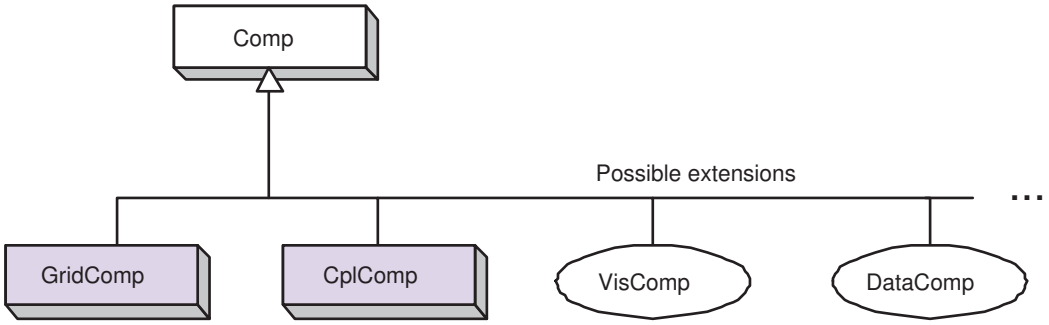


Figure 6: An `ESMF_StateReconcile()` call creates proxy objects for use in subsequent communication calls. The reconcile call would normally be made during Coupler initialization.



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.

```
  ------------------------------------------------------------------------------
  ------------------------------------------------------------------------------
   EXAMPLE:  This is an AppDriver.F90 file for a sequential ESMF application.
  ------------------------------------------------------------------------------
  ------------------------------------------------------------------------------

    The ChangeMe.F90 file that's included below contains a number of
    definitions that are used by the AppDriver, such as the name of the
    application's main configuration file and the name of the application's
    SetServices routine.  This file is in the same directory as the
    AppDriver.F90 file.
  ------------------------------------------------------------------------------

 #include "ChangeMe.F90"

     program ESMF_AppDriver
 #define ESMF_METHOD "program ESMF_AppDriver"

 #include "ESMF.h"

     ! ESMF module, defines all ESMF data types and procedures
     use ESMF

     ! Gridded Component registration routines.  Defined in "ChangeMe.F90"
     use USER_APP_Mod, only : SetServices => USER_APP_SetServices

     implicit none

  ------------------------------------------------------------------------------
```

```
  Define local variables
------------------------------------------------------------------------

   ! Components and States
   type(ESMF_GridComp) :: compGridded
   type(ESMF_State) :: defaultstate

   ! Configuration information
   type(ESMF_Config) :: config

   ! A common Grid
   type(ESMF_Grid) :: grid

   ! A Clock, a Calendar, and timesteps
   type(ESMF_Clock) :: clock
   type(ESMF_TimeInterval) :: timeStep
   type(ESMF_Time) :: startTime
   type(ESMF_Time) :: stopTime

   ! Variables related to the Grid
   integer :: i_max, j_max

   ! Return codes for error checks
   integer :: rc, localrc

------------------------------------------------------------------------
  Initialize ESMF.  Note that an output Log is created by default.
------------------------------------------------------------------------

   call ESMF_Initialize(defaultCalKind=ESMF_CALKIND_GREGORIAN, rc=localrc)
   if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
       ESMF_CONTEXT, rcToReturn=rc)) &
       call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

   call ESMF_LogWrite("ESMF AppDriver start", ESMF_LOGMSG_INFO)

------------------------------------------------------------------------
  Create and load a configuration file.
  The USER_CONFIG_FILE is set to sample.rc in the ChangeMe.F90 file.
  The sample.rc file is also included in the directory with the
  AppDriver.F90 file.
------------------------------------------------------------------------

   config = ESMF_ConfigCreate(rc=localrc)
   if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
       ESMF_CONTEXT, rcToReturn=rc)) &
       call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

   call ESMF_ConfigLoadFile(config, USER_CONFIG_FILE, rc = localrc)
   if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
       ESMF_CONTEXT, rcToReturn=rc)) &
       call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

------------------------------------------------------------------------
  Get configuration information.
```

```
  A configuration file like sample.rc might include:
  - size and coordinate information needed to create the default Grid.
  - the default start time, stop time, and running intervals
    for the main time loop.
------------------------------------------------------------------------

   call ESMF_ConfigGetAttribute(config, i_max, label='I Counts:', &
     default=10, rc=localrc)
   if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
       ESMF_CONTEXT, rcToReturn=rc)) &
       call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)
   call ESMF_ConfigGetAttribute(config, j_max, label='J Counts:', &
     default=40, rc=localrc)
   if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
       ESMF_CONTEXT, rcToReturn=rc)) &
       call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

------------------------------------------------------------------------
  Create the top Gridded Component.
------------------------------------------------------------------------

   compGridded = ESMF_GridCompCreate(name="ESMF Gridded Component", &
       rc=localrc)
   if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
       ESMF_CONTEXT, rcToReturn=rc)) &
       call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

   call ESMF_LogWrite("Component Create finished", ESMF_LOGMSG_INFO)

------------------------------------------------------------------------
  Register the set services method for the top Gridded Component.
------------------------------------------------------------------------

   call ESMF_GridCompSetServices(compGridded, userRoutine=SetServices, rc=rc)
   if (ESMF_LogFoundError(rc, msg="Registration failed", rcToReturn=rc)) &
       call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

------------------------------------------------------------------------
  Create and initialize a Clock.
------------------------------------------------------------------------

     call ESMF_TimeIntervalSet(timeStep, s=2, rc=localrc)
     if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
           ESMF_CONTEXT, rcToReturn=rc)) &
           call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

     call ESMF_TimeSet(startTime, yy=2004, mm=9, dd=25, rc=localrc)
     if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
           ESMF_CONTEXT, rcToReturn=rc)) &
           call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

     call ESMF_TimeSet(stopTime, yy=2004, mm=9, dd=26, rc=localrc)
     if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
           ESMF_CONTEXT, rcToReturn=rc)) &
```

```
          call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

     clock = ESMF_ClockCreate(timeStep, startTime, stopTime=stopTime, &
               name="Application Clock", rc=localrc)
     if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
           ESMF_CONTEXT, rcToReturn=rc)) &
           call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

------------------------------------------------------------------------
  Create and initialize a Grid.

  The default lower indices for the Grid are (/1,1/).
  The upper indices for the Grid are read in from the sample.rc file,
  where they are set to (/10,40/).  This means a Grid will be
  created with 10 grid cells in the x direction and 40 grid cells in the
  y direction.  The Grid section in the Reference Manual shows how to set
  coordinates.
------------------------------------------------------------------------

     grid = ESMF_GridCreateNoPeriDim(maxIndex=(/i_max, j_max/), &
                             name="source grid", rc=localrc)
     if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
           ESMF_CONTEXT, rcToReturn=rc)) &
           call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

     ! Attach the grid to the Component
     call ESMF_GridCompSet(compGridded, grid=grid, rc=localrc)
     if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
           ESMF_CONTEXT, rcToReturn=rc)) &
           call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

------------------------------------------------------------------------
  Create and initialize a State to use for both import and export.
  In a real code, separate import and export States would normally be
  created.
------------------------------------------------------------------------

     defaultstate = ESMF_StateCreate(name="Default State", rc=localrc)
     if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
           ESMF_CONTEXT, rcToReturn=rc)) &
           call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

------------------------------------------------------------------------
  Call the initialize, run, and finalize methods of the top component.
  When the initialize method of the top component is called, it will in
  turn call the initialize methods of all its child components, they
  will initialize their children, and so on.  The same is true of the
  run and finalize methods.
------------------------------------------------------------------------

     call ESMF_GridCompInitialize(compGridded, importState=defaultstate, &
       exportState=defaultstate, clock=clock, rc=localrc)
     if (ESMF_LogFoundError(rc, msg="Initialize failed", rcToReturn=rc)) &
         call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)
```

```
      call ESMF_GridCompRun(compGridded, importState=defaultstate, &
        exportState=defaultstate, clock=clock, rc=localrc)
      if (ESMF_LogFoundError(rc, msg="Run failed", rcToReturn=rc)) &
          call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

      call ESMF_GridCompFinalize(compGridded, importState=defaultstate, &
        exportState=defaultstate, clock=clock, rc=localrc)
      if (ESMF_LogFoundError(rc, msg="Finalize failed", rcToReturn=rc)) &
          call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)


!------------------------------------------------------------------------
  Destroy objects.
!------------------------------------------------------------------------

      call ESMF_ClockDestroy(clock, rc=localrc)
      if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
        ESMF_CONTEXT, rcToReturn=rc)) &
        call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

      call ESMF_StateDestroy(defaultstate, rc=localrc)
      if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
        ESMF_CONTEXT, rcToReturn=rc)) &
        call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

      call ESMF_GridCompDestroy(compGridded, rc=localrc)
      if (ESMF_LogFoundError(localrc, ESMF_ERR_PASSTHRU, &
        ESMF_CONTEXT, rcToReturn=rc)) &
        call ESMF_Finalize(rc=localrc, endflag=ESMF_END_ABORT)

!------------------------------------------------------------------------
  Finalize and clean up.
!------------------------------------------------------------------------

    call ESMF_Finalize()

    end program ESMF_AppDriver
```

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 ESMF_Initialize - Initialize ESMF

INTERFACE:

```
subroutine ESMF_Initialize(configFilenameFromArgNum, &
  configFilename, configKey, &
  defaultDefaultCalKind, defaultCalKind, &
  defaultDefaultLogFilename, defaultLogFilename, &
  defaultLogAppendFlag, logAppendFlag, defaultLogKindFlag, logKindFlag, &
  mpiCommunicator,  ioUnitLBound, ioUnitUBound, &
  defaultGlobalResourceControl, globalResourceControl, config, hconfig, &
  vm, rc)
```

ARGUMENTS:

```
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer,                  intent(in),  optional :: configFilenameFromArgNum
      character(len=*),         intent(in),  optional :: configFilename
      character(len=*),         intent(in),  optional :: configKey(:)
      type(ESMF_CalKind_Flag), intent(in),  optional :: defaultDefaultCalKind
      type(ESMF_CalKind_Flag), intent(in),  optional :: defaultCalKind
      character(len=*),         intent(in),  optional :: defaultDefaultLogFilename
      character(len=*),         intent(in),  optional :: defaultLogFilename
      logical,                  intent(in),  optional :: defaultLogAppendFlag
      logical,                  intent(in),  optional :: logAppendFlag
      type(ESMF_LogKind_Flag), intent(in),  optional :: defaultLogKindFlag
      type(ESMF_LogKind_Flag), intent(in),  optional :: logKindFlag
      integer,                  intent(in),  optional :: mpiCommunicator
      integer,                  intent(in),  optional :: ioUnitLBound
      integer,                  intent(in),  optional :: ioUnitUBound
      logical,                  intent(in),  optional :: defaultGlobalResourceControl
      logical,                  intent(in),  optional :: globalResourceControl
      type(ESMF_Config),        intent(out), optional :: config
      type(ESMF_HConfig),       intent(out), optional :: hconfig
      type(ESMF_VM),            intent(out), optional :: vm
      integer,                  intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

- This interface was modified since ESMF version 5.2.0r. The fact that code using this interface compiles with the current ESMF version does not guarantee that it compiles with previous versions of this interface. If user code compatibility with version 5.2.0r is desired then care must be taken to limit the use of this interface to features that were available in the 5.2.0r release.
  Changes made after the 5.2.0r release:

  7.0.0 Added argument `logAppendFlag` to allow specifying that the existing log files will be overwritten.

  8.2.0 Added argument `globalResourceControl` to support ESMF-aware threading and resource control on the global VM level.
  Added argument `config` to return default handle to the defaultConfig.
  Renamed argument `defaultConfigFilename` to `configFilename`, in order to clarify that provided settings in the Config file are *not* defaults, but final overrides.
  Introduce `default` prefixed arguments: `defaultDefaultLogFilename`, `defaultLogAppendFlag`, `defaultLogKindFlag`, `defaultGlobalResourceControl`. These arguments allow specification of defaults for the associated settings. This default can be overridden via the associated argument, without the extra `default` prefix, either specified in the call, or within the specified Config file.

  8.5.0 Added argument `configKey` to support custom location of the map of predefined initialization options for YAML configurations.
  Added argument `configFilenameFromArgNum` to support config file specification via the command line.

  8.6.0 Added `defaultDefaultCalKind` argument to allow specifiation of a default for `defaultCalKind`.

  8.7.0 Added argument `hconfig` to simplify direct access to the default `ESMF_HConfig` object.

DESCRIPTION:

This method must be called once on each PET before any other ESMF methods are used. The method contains a barrier before returning, ensuring that all processes made it successfully through initialization.

Typically `ESMF_Initialize()` will call `MPI_Init()` internally unless MPI has been initialized by the user code before initializing the framework. If the MPI initialization is left to `ESMF_Initialize()` it inherits all of the MPI implementation dependent limitations of what may or may not be done before `MPI_Init()`. For instance, it is unsafe for some MPI implementations, such as MPICH1, to do I/O before the MPI environment is initialized. Please consult the documentation of your MPI implementation for details.

Note that when using MPICH1 as the MPI library, ESMF needs to use the application command line arguments for `MPI_Init()`. However, ESMF acquires these arguments internally and the user does not need to worry about providing them. Also, note that ESMF does not alter the command line arguments, so that if the user obtains them they will be as specified on the command line (including those which MPICH1 would normally strip out).

`ESMF_Initialize()` supports running ESMF inside a user MPI program. Details of this feature are discussed under the VM example **??**. It is not necessary that all MPI ranks are handed to ESMF. Section **??** shows how an MPI communicator can be used to execute ESMF on a subset of MPI ranks. `ESMF_Initialize()` supports running multiple concurrent instances of ESMF under the same user MPI program. This feature is discussed under **??**.

In order to use any of the advanced resource management functions that ESMF provides via the `ESMF_*CompSetVM*()` methods, the MPI environment must be thread-safe. `ESMF_Initialize()` handles this automatically if it is in charge of initializing MPI. However, if the user code initializes MPI before calling into `ESMF_Initialize()`, it must do so via `MPI_Init_thread()`, specifying `MPI_THREAD_SERIALIZED` or above for the required level of thread support.

In cases where `ESMF_*CompSetVM*()` methods are used to move processing elements (PEs), i.e. CPU cores, between persistent execution threads (PETs), ESMF uses POSIX signals between PETs. In order to do so safely, the proper signal handlers must be installed *before* MPI is initialized. `ESMF_Initialize()` handles this automatically

if it is in charge of initializing MPI. If, however, MPI is explicitly initialized by user code, then to ensure correct signal handling it is necessary to call ESMF_InitializePreMPI() from the user code prior to the MPI initialization.

By default, ESMF_Initialize() opens multiple error log files, one per processor. This is very useful for debugging purpose. However, when running the application on a large number of tasks, opening a large number of log files and writing log messages from all the tasks can become a performance bottleneck. Therefore, it is recommended for production runs to set logKindFlag to ESMF_LOGKIND_NONE, or ESMF_LOGKIND_Multi_On_Error. The latter only creates log files when an error occurs.

When integrating ESMF with applications where Fortran unit number conflicts exist, the optional ioUnitLBound and ioUnitUBound arguments may be used to specify an alternate unit number range. See section **??** for more information on how ESMF uses Fortran unit numbers.

Before exiting the application the user must call ESMF_Finalize() to release resources and clean up ESMF gracefully. See the ESMF_Finalize() documentation about details relating to the MPI environment.

The arguments are:

[configFilenameFromArgNum] Index of the command line argument specifying the config file name. If the specified command line argument does not exist, or configFilenameFromArgNum was not specified, the configFilename argument, if provided, is used by default.

[configFilename] Name of the configuration file for the entire application. If this argument is specified, the configuration file must exist. Its content is read during ESMF_Initialize(), and returned in optional argument config if present.

The traditional ESMF_Config format and the YAML format are supported. The latter is identified by file suffix .yaml and .yml, including all lower/upper case letter combinations that map to either suffix.

In the case of the traditional ESMF_Config format, the predefined labels of initialization options discussed below are expected on the top level of the configuration. The expected termination character for this case is a single colon following each label.

For the YAML case, the predefined initialization option labels are expected as the keys of a map. If the optional argument configKey is specified, it is used to locate this map. The map is expected as the terminal value of a succession of mappings:

```
configKey(1) :
  configKey(2) :
    ...
      configKey(size(configKey)) :
        {map of specified init options}
```

By default, in the absence of argument configKey, the top level itself is searched for a mapping of predefined labels, analogous to the traditional case.

If any of the following predefined labels are found in the specified configuration file (as per the above defined rules), their *values* are used to set the associated ESMF_Initialize() argument, overriding any defaults. If the same argument is also specified in the ESMF_Initialize() call directly, an error is returned, and ESMF is not initialized. The supported config labels are:

- defaultCalKind
- defaultLogFilename
- logAppendFlag
- logKindFlag
- globalResourceControl

ESMF allows the user to affect certain details about the execution of an application through a number of runtime environment variables. The following list of variables are checked within the specified configuration file. If a matching label is found, the respective value is set, potentially overriding the value defined within the user environment for the same variable.

- ESMF_RUNTIME_PROFILE
- ESMF_RUNTIME_PROFILE_OUTPUT
- ESMF_RUNTIME_PROFILE_PETLIST
- ESMF_RUNTIME_TRACE
- ESMF_RUNTIME_TRACE_CLOCK
- ESMF_RUNTIME_TRACE_PETLIST
- ESMF_RUNTIME_TRACE_COMPONENT
- ESMF_RUNTIME_TRACE_FLUSH
- ESMF_RUNTIME_COMPLIANCECHECK

[configKey] If present, use configKey to find the map of predefined initialization options that are used during ESMF initialization. The default is to search the top level of the configuration for the labels directly. The configKey option is only supported for YAML configurations. An error is returned if configKey is specified for the traditional ESMF_Config case.

[defaultDefaultCalKind] Default value for argument defaultCalKind, the calendar used by ESMF Time Manger by default. If not specified, defaults to ESMF_CALKIND_NOCALENDAR.

[defaultCalKind] Sets the default calendar to be used by ESMF Time Manager. See section **??** for a list of valid options. If not specified, defaults according to defaultDefaultCalKind.

[defaultDefaultLogFilename] Default value for argument defaultLogFilename, the name of the default log file for warning and error messages. If not specified, the default is ESMF_LogFile.

[defaultLogFilename] Name of the default log file for warning and error messages. If not specified, defaults according to defaultDefaultLogFilename.

[defaultLogAppendFlag] Default value for argument logAppendFlag, indicating the overwrite behavior in case the default log file already exists. If not specified, the default is .true..

[logAppendFlag] If the default log file already exists, a value of .false. will set the file position to the beginning of the file. A value of .true. sets the position to the end of the file. If not specified, defaults according to defaultLogAppendFlag.

[defaultLogKindFlag] Default value for argument logKindFlag, setting the LogKind of the default ESMF log. If not specified, the default is ESMF_LOGKIND_MULTI.

[logKindFlag] Sets the LogKind of the default ESMF log. See section **??** for a list of valid options. If not specified, defaults according to defaultLogKindFlag.

[mpiCommunicator] MPI communicator defining the group of processes on which the ESMF application is running. See section **??** and **??** for details. If not specified, defaults to MPI_COMM_WORLD.

[ioUnitLBound] Lower bound for Fortran unit numbers used within the ESMF library. Fortran units are primarily used for log files. Legal unit numbers are positive integers. A value higher than 10 is recommended in order to avoid the compiler-specific reservations which are typically found on the first few units. If not specified, defaults to ESMF_LOG_FORT_UNIT_NUMBER, which is distributed with a value of 50.

[ioUnitUBound] Upper bound for Fortran unit numbers used within the ESMF library. Must be set to a value at least 5 units higher than ioUnitLBound. If not specified, defaults to ESMF_LOG_UPPER, which is distributed with a value of 99.

[defaultGlobalResourceControl] Default value for argument `globalResourceControl`, indicating whether PETs of the global VM are pinned to PEs and the OpenMP threading level is reset. If not specified, the default is `.false.`.

[globalResourceControl] For `.true.`, each global PET is pinned to the corresponding PE (i.e. CPU core) in order. Further, if OpenMP support is enabled for the ESMF installation (during build time), the `OMP_NUM_THREADS` is set to `1` on every PET, regardless of the setting in the launching environment. The `.true.` setting is recommended for applications that utilize the ESMF-aware threading and resource control features. For `.false.`, global PETs are *not* pinned by ESMF, and `OMP_NUM_THREADS` is *not* modified. If not specified, defaults according to `defaultGlobalResourceControl`.

[config] Returns the default `ESMF_Config` if the `configFilename` argument was provided. Otherwise the presence of this argument triggers an error.

[hconfig] Returns the default `ESMF_HConfig` if the `configFilename` argument was provided. Otherwise the presence of this argument triggers an error.

[vm] Returns the global `ESMF_VM` that was created during initialization.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.4.2 ESMF_InitializePreMPI - Initialize parts of ESMF that must happen before MPI is initialized

INTERFACE:

```
subroutine ESMF_InitializePreMPI(rc)
```

ARGUMENTS:

```
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,                  intent(out), optional :: rc
```

DESCRIPTION:

This method is *only* needed for cases where MPI is initialized explicitly by user code. In most typical cases `ESMF_Initialize()` is called before MPI is initialized, and takes care of all the internal initialization, including MPI.

There are circumstances where it is necessary or convenient to initialize MPI before calling into `ESMF_Initialize()`. This option is supported by ESMF, and for most cases no special action is required on the user side. However, for cases where `ESMF_*CompSetVM*()` methods are used to move processing elements (PEs), i.e. CPU cores, between persistent execution threads (PETs), ESMF uses POSIX signals between PETs. In order to do so safely, the proper signal handlers must be installed before MPI is initialized. This is accomplished by calling `ESMF_InitializePreMPI()` from the user code prior to the MPI initialization.

Note also that in order to use any of the advanced resource management functions that ESMF provides via the `ESMF_*CompSetVM*()` methods, the MPI environment must be thread-safe. `ESMF_Initialize()` handles this automatically if it is in charge of initializing MPI. However, if the user code initializes MPI before calling into `ESMF_Initialize()`, it must do so via `MPI_Init_thread()`, specifying `MPI_THREAD_SERIALIZED` or above for the required level of thread support.

The arguments are:

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.4.3 ESMF_IsInitialized - Query Initialized status of ESMF

INTERFACE:

```
function ESMF_IsInitialized(rc)
```

RETURN VALUE:

```
logical :: ESMF_IsInitialized
```

ARGUMENTS:

```
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
   integer,                intent(out), optional :: rc
```

DESCRIPTION:

Returns `.true.` if the framework has been initialized. This means that `ESMF_Initialize()` has been called. Otherwise returns `.false.`. If an error occurs, i.e. `rc /= ESMF_SUCCESS` is returned, the return value of the function will also be `.false.`.

The arguments are:

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.4.4 ESMF_IsFinalized - Query Finalized status of ESMF

INTERFACE:

```
function ESMF_IsFinalized(rc)
```

RETURN VALUE:

```
logical :: ESMF_IsFinalized
```

ARGUMENTS:

```
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
   integer,                intent(out), optional :: rc
```

DESCRIPTION:

Returns `.true.` if the framework has been finalized. This means that `ESMF_Finalize()` has been called. Otherwise returns `.false.`. If an error occurs, i.e. `rc /= ESMF_SUCCESS` is returned, the return value of the function will also be `.false.`.

The arguments are:

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.4.5 ESMF_Finalize - Clean up and shut down ESMF

INTERFACE:

```
subroutine ESMF_Finalize(endflag, rc)
```

ARGUMENTS:

```
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
   type(ESMF_End_Flag), intent(in), optional  :: endflag
   integer,             intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.2.0r. If code using this interface compiles with any version of ESMF starting with 5.2.0r, then it will compile with the current version.

DESCRIPTION:

This must be called once on each PET before the application exits to allow ESMF to flush buffers, close open connections, and release internal resources cleanly. The optional argument `endflag` may be used to indicate the mode of termination. Note that this call must be issued only once per PET with `endflag=ESMF_END_NORMAL`, and that this call may not be followed by `ESMF_Initialize()`. This last restriction means that it is not possible to restart ESMF within the same execution.

The arguments are:

[endflag] Specify mode of termination. The default is `ESMF_END_NORMAL` which waits for all PETs of the global VM to reach `ESMF_Finalize()` before termination. See section 16.2.1 for a complete list and description of valid flags.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

16.4.6 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 17.2.1 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.

16.4.7 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 17.2.1 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 17.2.2, 17.2.3, 17.2.4, and **??**, **??**, **??** for examples of how to write entry points for the standard ESMF Component methods.

16.4.8 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.

16.4.9 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 **??**.

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.

17.2.1 Implement a user-code SetServices routine

Every ESMF_GridComp is required to provide and document a public set services routine. It can have any name, but must follow the declaration below: a subroutine which takes an ESMF_GridComp 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 set services routine must call the ESMF method ESMF_GridCompSetEntryPoint() to register with the framework what user-code subroutines should be called to initialize, run, and finalize the component. There are additional routines which can be registered as well, for checkpoint and restart functions.

Note that the actual subroutines being registered do not have to be public to this module; only the set services routine itself must be available to be used by other code.

```
! Example Gridded Component
module ESMF_GriddedCompEx

! ESMF Framework module
use ESMF
implicit none
public GComp_SetServices
public GComp_SetVM

contains

subroutine GComp_SetServices(comp, rc)
  type(ESMF_GridComp)   :: comp   ! must not be optional
  integer, intent(out)  :: rc     ! must not be optional

  ! Set the entry points for standard ESMF Component methods
  call ESMF_GridCompSetEntryPoint(comp, ESMF_METHOD_INITIALIZE, &
                            userRoutine=GComp_Init, rc=rc)
  call ESMF_GridCompSetEntryPoint(comp, ESMF_METHOD_RUN, &
```

```
                              userRoutine=GComp_Run, rc=rc)
  call ESMF_GridCompSetEntryPoint(comp, ESMF_METHOD_FINALIZE, &
                              userRoutine=GComp_Final, rc=rc)

  rc = ESMF_SUCCESS

end subroutine
```

17.2.2 Implement a user-code `Initialize` routine

When a higher level component is ready to begin using an `ESMF_GridComp`, it will call its initialize routine.

The component writer must supply a subroutine with the exact interface shown below. Arguments must not be declared as optional, and the types and order must match.

At initialization time the component can allocate data space, open data files, set up initial conditions; anything it needs to do to prepare to run.

The `rc` return code should be set if an error occurs, otherwise the value `ESMF_SUCCESS` should be returned.

```
subroutine GComp_Init(comp, importState, exportState, clock, rc)
  type(ESMF_GridComp)   :: comp                  ! must not be optional
  type(ESMF_State)      :: importState           ! must not be optional
  type(ESMF_State)      :: exportState           ! must not be optional
  type(ESMF_Clock)      :: clock                 ! must not be optional
  integer, intent(out)  :: rc                    ! must not be optional

  print *, "Gridded Comp Init starting"

  ! This is where the model specific setup code goes.

  ! If the initial Export state needs to be filled, do it here.
  !call ESMF_StateAdd(exportState, field, rc)
  !call ESMF_StateAdd(exportState, bundle, rc)
  print *, "Gridded Comp Init returning"

  rc = ESMF_SUCCESS

end subroutine GComp_Init
```

17.2.3 Implement a user-code `Run` routine

During the execution loop, the run routine may be called many times. Each time it should read data from the `importState`, use the `clock` to determine what the current time is in the calling component, compute new values or process the data, and produce any output and place it in the `exportState`.

When a higher level component is ready to use the `ESMF_GridComp` it will call its run routine.

The component writer must supply a subroutine with the exact interface shown below. Arguments must not be declared as optional, and the types and order must match.

It is expected that this is where the bulk of the model computation or data analysis will occur.

The `rc` return code should be set if an error occurs, otherwise the value `ESMF_SUCCESS` should be returned.

```
subroutine GComp_Run(comp, importState, exportState, clock, rc)
  type(ESMF_GridComp)   :: comp                  ! must not be optional
  type(ESMF_State)      :: importState           ! must not be optional
  type(ESMF_State)      :: exportState           ! must not be optional
  type(ESMF_Clock)      :: clock                 ! must not be optional
  integer, intent(out)  :: rc                    ! must not be optional

  print *, "Gridded Comp Run starting"
  ! call ESMF_StateGet(), etc to get fields, bundles, arrays
  !  from import state.

  ! This is where the model specific computation goes.

  ! Fill export state here using ESMF_StateAdd(), etc

  print *, "Gridded Comp Run returning"

  rc = ESMF_SUCCESS

end subroutine GComp_Run
```

17.2.4 Implement a user-code `Finalize` routine

At the end of application execution, each `ESMF_GridComp` should deallocate data space, close open files, and flush final results. These functions should be placed in a finalize routine.

The component writer must supply a subroutine with the exact interface shown below. Arguments must not be declared as optional, and the types and order must match.

The `rc` return code should be set if an error occurs, otherwise the value `ESMF_SUCCESS` should be returned.

```
subroutine GComp_Final(comp, importState, exportState, clock, rc)
  type(ESMF_GridComp)   :: comp                  ! must not be optional
  type(ESMF_State)      :: importState           ! must not be optional
  type(ESMF_State)      :: exportState           ! must not be optional
  type(ESMF_Clock)      :: clock                 ! must not be optional
  integer, intent(out)  :: rc                    ! must not be optional

  print *, "Gridded Comp Final starting"

  ! Add whatever code here needed

  print *, "Gridded Comp Final returning"

  rc = ESMF_SUCCESS

end subroutine GComp_Final
```
