Earth System Modeling Framework

ESMF Reference Manual for Fortran

Version 8.2.0 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, 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

October 21, 2021

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.

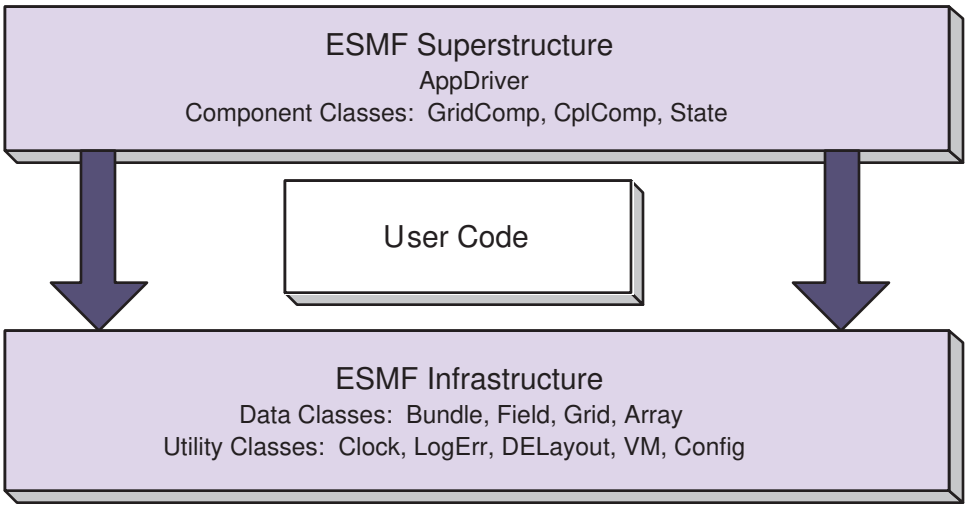
2 The ESMF Reference Manual for Fortran

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

The 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 the method in which they were declared has returned.

An exception to this is when a shallow object, such as a Time, is stored in a deep object such as a Clock. The Clock then carries a copy of the Time in persistent memory. The Time 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 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.

6.4 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.

6.5 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.

6.6 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.

6.7 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.

6.8 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 24.2). Regridding, also called remapping or interpolation, is the process of changing the grid that underlies data values while preserving qualities of the original data. Different kinds of transformations are appropriate for different problems. Regridding may be needed when communicating data between Earth system model components such as land and atmosphere, or between different data sets to support operations such as visualization.

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

There are two options for accessing ESMF regridding functionality: integrated and offline. Integrated regridding is a process whereby interpolation weights are generated via subroutine calls during the execution of the user's code. The integrated regridding can also perform the parallel sparse matrix multiplication. In other words, ESMF integrated regridding allows a user to perform the whole process of interpolation within their code. For a further description of ESMF integrated regridding please see Section 26.3.25. 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.

In the latest ESMF implementation, the `ESMF_RegridWeightGen` command line toll can detect the type of the input grid files automatically. The user doesn't need to provide the source and destination grid file type arguments anymore. The following arguments `-t`, `-src_type`, `-dst_type`, `-src_meshname`, and `-dst_meshname` are no longer needed. If provided, the application will simply ingore them.

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()` 31.3.8 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()` 31.3.12. Such a grid is stored in the GRIDSPEC Mosaic and tile file format. 12.8.5 The method `ESMF_MeshCreate()` **??** is used to create an ESMF_Mesh object, if the source or destination grid is an unstructured grid. When making this call, the flag `convert3D` is set to `TRUE` to convert the 2D coordinates into 3D Cartesian coordinates. Internally `ESMF_FieldRegridStore()` is used to generate the weight table and indices table representing the interpolation matrix.

12.2 Regridding Options

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

12.2.1 Poles

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

12.2.2 Masking

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

missing values defined in the data variable. The regridding application provides the argument "−−src_missingvalue" or "−−dst_missingvalue" for users to specify the variable name from where the mask can be constructed.

12.2.3 Extrapolation

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

12.2.4 Unmapped destination points

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

12.2.5 Line type

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

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

12.3 Regridding Methods

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

12.3.1 Bilinear

The default interpolation method for the weight generation application is bilinear. The algorithm used by this application to generate the bilinear weights is the standard one found in many textbooks. Each destination point is mapped

to a location in the source Mesh, the position of the destination point relative to the source points surrounding it is used to calculate the interpolation weights. A restriction on bilinear interpolation is that ESMF doesn't support self-intersecting cells (e.g. a cell twisted into a bow tie) in the source grid.

12.3.2 Patch

This application can also be used to generate patch interpolation weights. Patch interpolation is the ESMF version of a technique called "patch recovery" commonly used in finite element modeling [**?**] [**?**]. 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 24.2.9.

12.3.5 Second-order conservative

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

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

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

12.4 Conservation

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

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

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

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

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

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

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

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

For weights generated using fraction area normalization (set by specifying "--norm_type fracarea"), no adjustment of the destination field (`dst_field`) by the destination fraction is necessary. The following pseudo-code shows how to compute the total destination integral (`dst_total`) given the destination field values (`dst_field`) resulting from the sparse matrix multiplication of the weights in the weight file by the source field, the destination area (`dst_area`) called `area_b` in the weight file, and the destination fraction (`dst_frac`) called `frac_b` in the weight file:

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

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

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

12.6 Usage

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

```
ESMF_RegridWeightGen
        --source|-s src_grid_filename
        --destination|-d dst_grid_filename
        --weight|-w out_weight_file
        [--method|-m bilinear|patch|nearestdtos|neareststod|conserve|conserve2nd]
        [--pole|-p none|all|teeth|1|2|..]
        [--line_type|-l cartesian|greatcircle]
        [--norm_type dstarea|fracarea]
        [--extrap_method none|neareststod|nearestidavg|nearestd|creep|creepnrstd]
        [--extrap_num_src_pnts <N>]
        [--extrap_dist_exponent <P>]
        [--extrap_num_levels <L>]
        [--ignore_unmapped|-i]
        [--ignore_degenerate]
        [-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]
        [--no_log]
        [--help]
        [--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
                      nearestdtos  - for nearest destination to source 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 normalization 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.

-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.

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

 --help           - 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 TILE	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

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:

```
  2.0   7 ------- 8 ------- 9
        |         |         |
        |    4    |    5    |
        |         |         |
  1.0   4 ------- 5 ------- 6
```

```
        |         |  \   3  |
        |    1    |    \    |
        |         |  2   \  |
  0.0   1 ------- 2 ------- 3

       0.0       1.0       2.0

        Node indices at corners
       Element indices in centers

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

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 `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 re-

quired. 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 24.2). Regridding, also called remapping or interpolation, is the process of changing

the grid that underlies data values while preserving qualities of the original data. Different kinds of transformations are appropriate for different problems. Regridding may be needed when communicating data between Earth system model components such as land and atmosphere, or between different data sets to support operations such as visualization.

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

The `ESMF_RegridWeightGen` command line tool described in Section 12 performs the first stage of the regridding process - generate the interpolation weight matrix. This tool not only calculates the interpolation weights, it also applies the weights to a list of variables stored in the source grid file and produces the interpolated values on the destination grid. The interpolated output variable is written out to the destination grid file. This tool supports three CF compliant file formats: the CF Single Tile grid file format(12.8.3) for a logically rectangular grid, the UGRID file format(12.8.4) for unstructured grid and the GRIDSPEC Mosaic file format(12.8.5) for cubed-sphere grid. For the GRIDSPEC Mosaic file format, the data are stored in seperate 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 seperate 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 varialbe does not exist in the destination grid file, a new variable will be created with the same type and matching dimensions as

the source variable. All the attributes of the source variable will be copied to the destination variable except those related to the grid definition (i.e. `coordinates` attribute if the destination file is in GRIDSPEC or MOSAIC format or `mesh` and `location` attributes if the destination file is in UGRID format.

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

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

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

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

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

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

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

13.2 Usage

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

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

  --srcdatafile       - 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             - 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

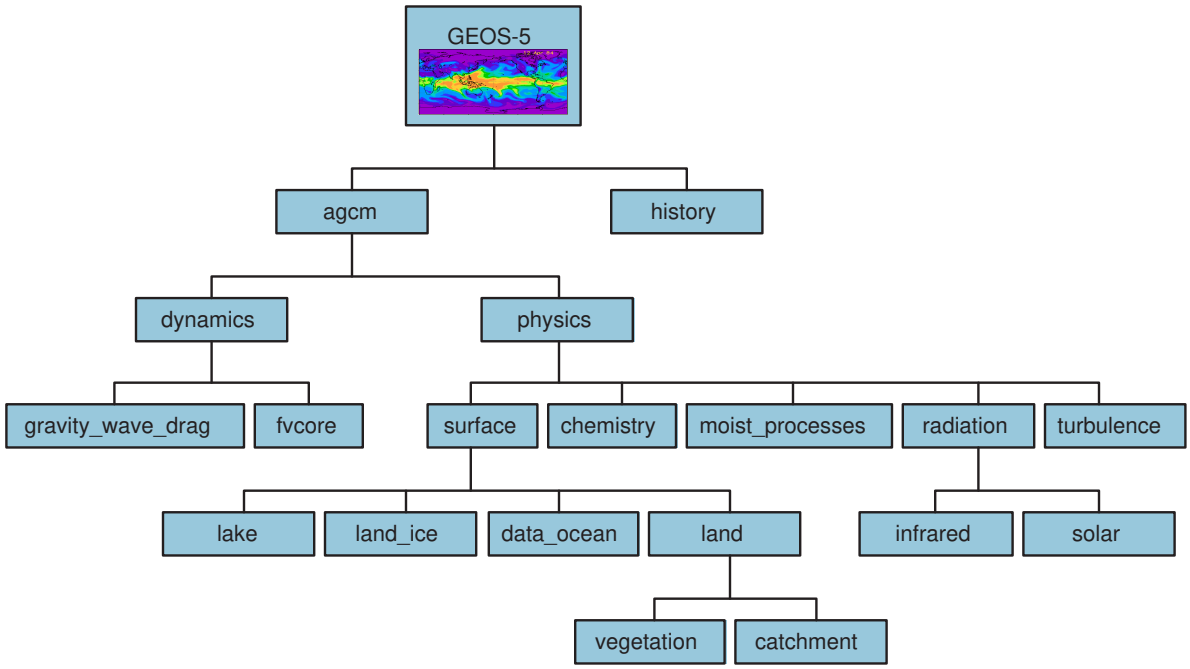
There are a small number of classes in the ESMF superstructure:

- **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.



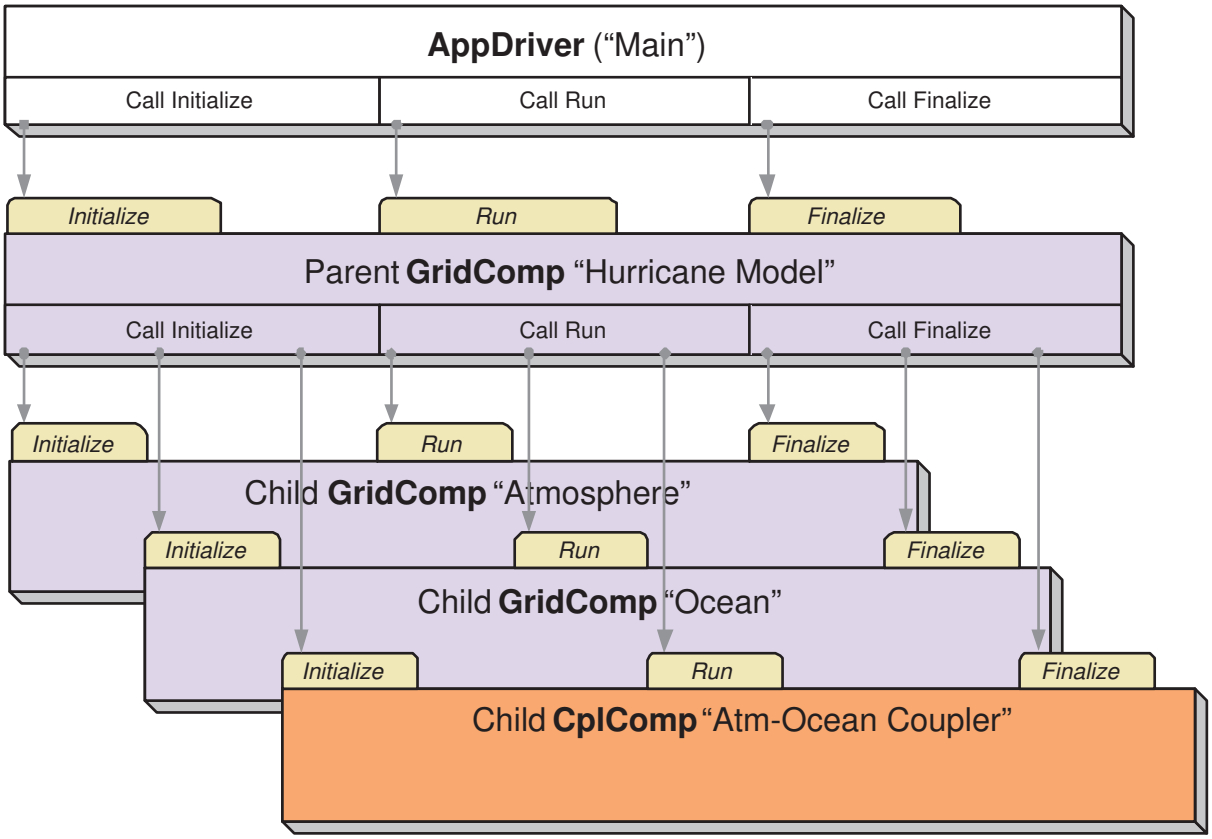
An ESMF coupled application typically involves a parent Gridded Component, two or more child Gridded Components and one or more Coupler Components.

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.

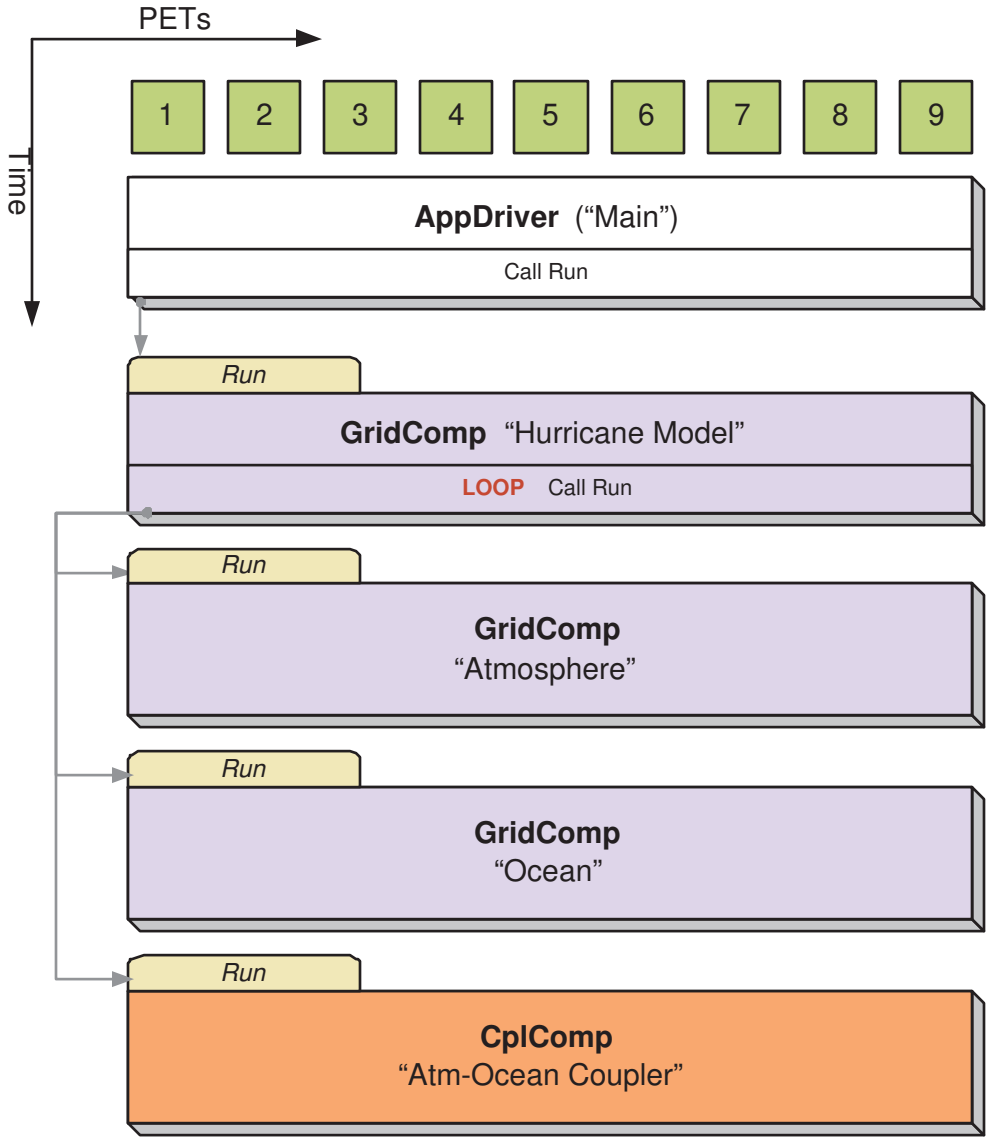


Figure 5: Schematic of the run method of a coupled application, with an “Atmosphere” and an “Ocean” Gridded Component running concurrently with an “Atm-Ocean Coupler.” The top-level “Hurricane Model” Gridded Component contains the sequencing information and time advancement loop. The application driver,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

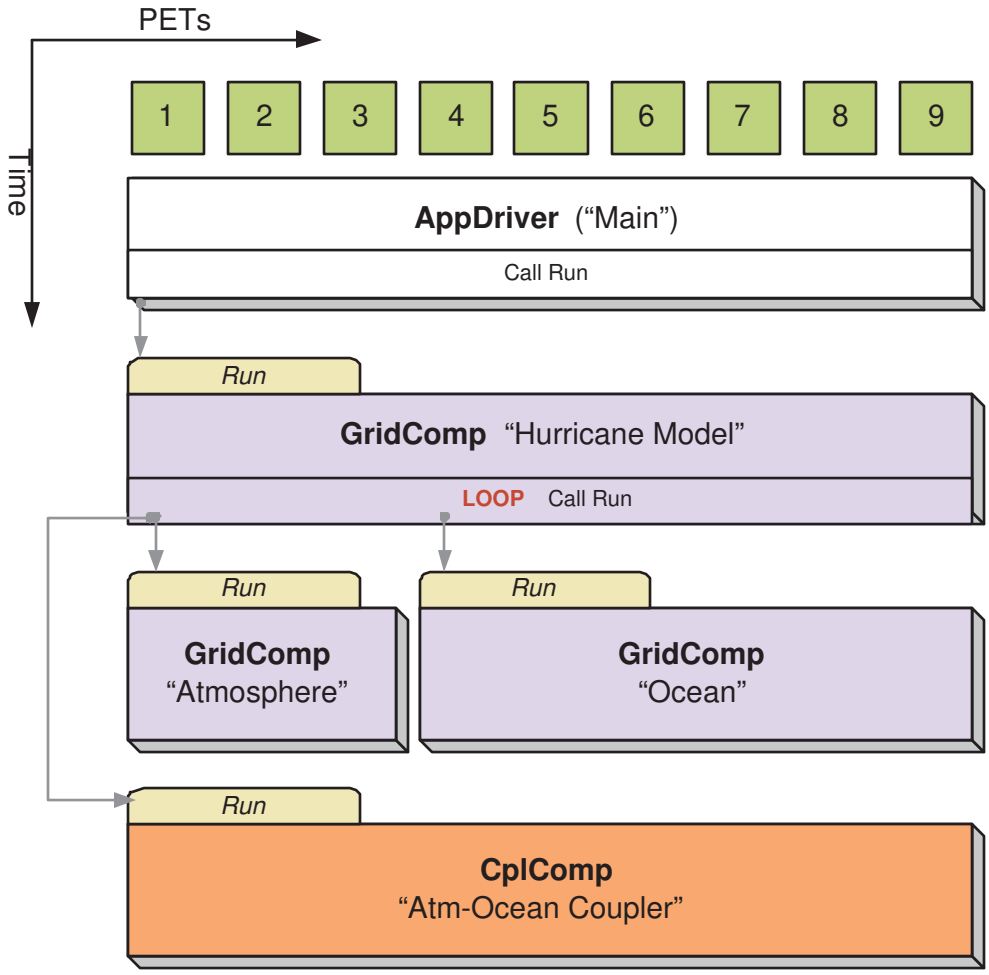
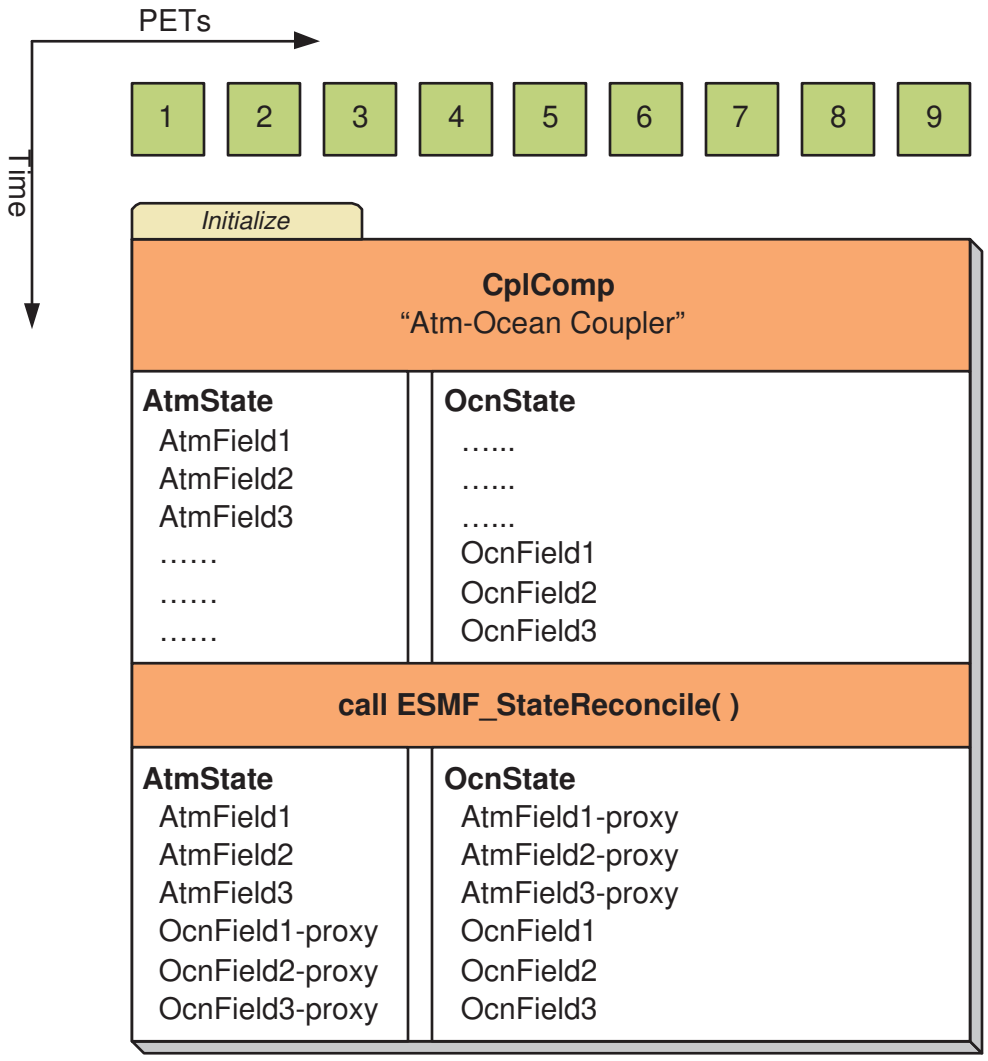
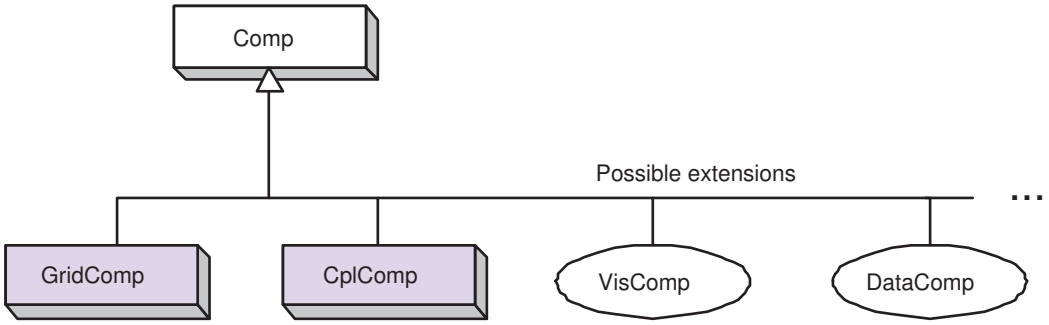


Figure 6: An `ESMF_StateReconcile()` call creates proxy objects for use in subsequent communication calls. The reconcile call would normally be made during Coupler initialization.



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(configFilename, &
  defaultCalKind, defaultDefaultLogFilename, defaultLogFilename, &
  defaultLogAppendFlag, logAppendFlag, defaultLogKindFlag, logKindFlag, &
  mpiCommunicator,  ioUnitLBound, ioUnitUBound, &
  defaultGlobalResourceControl, globalResourceControl, config, vm, rc)
```

ARGUMENTS:

```
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    character(len=*),        intent(in),  optional :: configFilename
    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_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.

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 MPICH, to do I/O before the MPI environment is initialized. Please consult the documentation of your MPI implementation for details.

Note that when using MPICH 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 MPICH 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()` will open multiple error log files, one per processor. This is very useful for debugging purpose. However, when running the application on a large number of processors, opening a large number of log files and writing log messages from all the processors could become a performance bottleneck. Therefore, it is recommended to turn the Error Log feature off in these situations by setting `logKindFlag` to ESMF_LOGKIND_NONE.

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:

[configFilename] Name of the configuration file for the entire application. If this argument is specified, the configuration file must exist, and its content is read during `ESMF_Initialize()`. If any of the following labels are found in the specified configuration file, 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:

- `defaultLogFilename:`
- `logAppendFlag:`
- `logKindFlag:`
- `globalResourceControl:`

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

[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.

[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 18.2.1 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 17.4.19 and 18.4.19, 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 17.4.19 and 18.4.19, 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 17.4.20 and 18.4.20, 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 18.2.1 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 18.2.2, 18.2.3, 18.2.4 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 17.2.5 and 18.2.5 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 21.1.

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

ESMF provides the Gridded Component derived type, `ESMF_GridComp`. An `ESMF_GridComp` must be created for every portion of the application that will be represented as a separate component. For example, in a climate model, there may be Gridded Components representing the land, ocean, sea ice, and atmosphere. If the application contains an ensemble of identical Gridded Components, every one has its own associated `ESMF_GridComp`. Each Gridded Component has its own name and is allocated a set of computational resources, in the form of an ESMF Virtual Machine, or `VM`.

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

17.2 Use and Examples

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

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

17.2.1 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
```

17.2.5 Implement a user-code `SetVM` routine

Every `ESMF_GridComp` can optionally provide and document a public set vm 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 vm routine is the only place where the child component can use the `ESMF_GridCompSetVMMaxPEs()`, or `ESMF_GridCompSetVMMaxThreads()`, or `ESMF_GridCompSetVMMinThreads()` call to modify aspects of its own VM.

A component's VM is started up right before its set services routine is entered. `ESMF_GridCompSetVM()` is executing in the parent VM, and must be called *before* `ESMF_GridCompSetServices()`.

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

  type(ESMF_VM) :: vm
  logical :: pthreadsEnabled

  ! Test for Pthread support, all SetVM calls require it
  call ESMF_VMGetGlobal(vm, rc=rc)
  call ESMF_VMGet(vm, pthreadsEnabledFlag=pthreadsEnabled, rc=rc)

  if (pthreadsEnabled) then
    ! run PETs single-threaded
    call ESMF_GridCompSetVMMinThreads(comp, rc=rc)
  endif

  rc = ESMF_SUCCESS

end subroutine

end module ESMF_GriddedCompEx
```

17.2.6 Set and Get the Internal State

ESMF provides the concept of an Internal State that is associated with a Component. Through the Internal State API a user can attach a private data block to a Component, and later retrieve a pointer to this memory allocation. Setting and getting of Internal State information are supported from anywhere in the Component's SetServices, Initialize, Run, or Finalize code.

The code below demonstrates the basic Internal State API of `ESMF_<Grid|Cpl>SetInternalState()` and `ESMF_<Grid|Cpl>GetInternalState()`. Notice that an extra level of indirection to the user data is necessary!

```
! ESMF Framework module
use ESMF
use ESMF_TestMod
implicit none

type(ESMF_GridComp) :: comp
integer :: rc, finalrc

! Internal State Variables
type testData
```

```
  sequence
    integer :: testValue
    real    :: testScaling
  end type

  type dataWrapper
  sequence
    type(testData), pointer :: p
  end type

  type(dataWrapper) :: wrap1, wrap2
  type(testData), target :: data
  type(testData), pointer :: datap  ! extra level of indirection


!------------------------------------------------------------------------

  call ESMF_Initialize(defaultlogfilename="InternalStateEx.Log", &
                    logkindflag=ESMF_LOGKIND_MULTI, rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

!------------------------------------------------------------------------

  !  Creation of a Component
  comp = ESMF_GridCompCreate(name="test", rc=rc)
  if (rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

!------------------------------------------------------------------------
! This could be called, for example, during a Component's initialize phase.

    ! Initialize private data block
  data%testValue = 4567
  data%testScaling = 0.5

  ! Set Internal State
  wrap1%p => data
  call ESMF_GridCompSetInternalState(comp, wrap1, rc)
  if (rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

!------------------------------------------------------------------------
! This could be called, for example, during a Component's run phase.

  ! Get Internal State
  call ESMF_GridCompGetInternalState(comp, wrap2, rc)
  if (rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE

  ! Access private data block and verify data
  datap => wrap2%p
  if ((datap%testValue .ne. 4567) .or. (datap%testScaling .ne. 0.5)) then
    print *, "did not get same values back"
    finalrc = ESMF_FAILURE
  else
    print *, "got same values back from GetInternalState as original"
  endif
```

When working with ESMF Internal States it is important to consider the applying scoping rules. The user must ensure that the private data block that is being referenced persists for the entire access period. This is not an issue in the previous example, where the private data block was defined on the scope of the main program. However, the Internal State construct is often useful inside of Component modules to hold Component specific data between calls. One option to ensure persisting private data blocks is to use the Fortran SAVE attribute either on local or module variables. A second option, illustrated in the following example, is to use Fortran pointers and user controlled memory management via allocate() and deallocate() calls.

One situation where the Internal State is useful is in the creation of ensembles of the same Component. In this case it can be tricky to distinguish which data, held in saved module variables, belongs to which ensemble member - especially if the ensemble members are executing on the same set of PETs. The Internal State solves this problem by providing a handle to instance specific data allocations.

```
module user_mod

  use ESMF

  implicit none

  ! module variables
  private

  ! Internal State Variables
  type testData
  sequence
    integer       :: testValue        ! scalar data
    real          :: testScaling      ! scalar data
    real, pointer :: testArray(:)     ! array data
  end type

  type dataWrapper
  sequence
    type(testData), pointer :: p
  end type



  contains !------------------------------------------------------------



  subroutine mygcomp_init(gcomp, istate, estate, clock, rc)
    type(ESMF_GridComp):: gcomp
    type(ESMF_State):: istate, estate
    type(ESMF_Clock):: clock
    integer, intent(out):: rc

    ! Local variables
    type(dataWrapper) :: wrap
    type(testData), pointer :: data
    integer :: i
```

```
  rc = ESMF_SUCCESS

  ! Allocate private data block
  allocate(data)

  ! Initialize private data block
  data%testValue = 4567         ! initialize scalar data
  data%testScaling = 0.5        ! initialize scalar data
  allocate(data%testArray(10))  ! allocate array data

  do i=1, 10
    data%testArray(i) = real(i) ! initialize array data
  enddo

  ! In a real ensemble application the initial data would be set to
  ! something unique for this ensemble member. This could be
  ! accomplished for example by reading a member specific config file
  ! that was specified by the driver code. Alternatively, Attributes,
  ! set by the driver, could be used to label the Component instances
  ! as specific ensemble members.

  ! Set Internal State
  wrap%p => data
  call ESMF_GridCompSetInternalState(gcomp, wrap, rc)

end subroutine !-------------------------------------------------------

subroutine mygcomp_run(gcomp, istate, estate, clock, rc)
  type(ESMF_GridComp):: gcomp
  type(ESMF_State):: istate, estate
  type(ESMF_Clock):: clock
  integer, intent(out):: rc

  ! Local variables
  type(dataWrapper) :: wrap
  type(testData), pointer :: data
  logical :: match = .true.
  integer :: i

  rc = ESMF_SUCCESS

  ! Get Internal State
  call ESMF_GridCompGetInternalState(gcomp, wrap, rc)
  if (rc/=ESMF_SUCCESS) return

  ! Access private data block and verify data
  data => wrap%p
  if (data%testValue .ne. 4567) match = .false.   ! test scalar data
  if (data%testScaling .ne. 0.5) match = .false.  ! test scalar data
  do i=1, 10
    if (data%testArray(i) .ne. real(i)) match = .false. ! test array data
  enddo

  if (match) then
```

```
      print *, "got same values back from GetInternalState as original"
    else
      print *, "did not get same values back"
      rc = ESMF_FAILURE
    endif

  end subroutine !-------------------------------------------------------

  subroutine mygcomp_final(gcomp, istate, estate, clock, rc)
    type(ESMF_GridComp):: gcomp
    type(ESMF_State):: istate, estate
    type(ESMF_Clock):: clock
    integer, intent(out):: rc

    ! Local variables
    type(dataWrapper) :: wrap
    type(testData), pointer :: data

    rc = ESMF_SUCCESS

    ! Get Internal State
    call ESMF_GridCompGetInternalState(gcomp, wrap, rc)
    if (rc/=ESMF_SUCCESS) return

    ! Deallocate private data block
    data => wrap%p
    deallocate(data%testArray)  ! deallocate array data
    deallocate(data)

  end subroutine !-------------------------------------------------------------


end module
```

17.3 Restrictions and Future Work

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

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

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

17.4 Class API

17.4.1 ESMF_GridCompAssignment(=) - GridComp assignment

INTERFACE:

```
interface assignment(=)
gridcomp1 = gridcomp2
```

ARGUMENTS:

```
type(ESMF_GridComp) :: gridcomp1
type(ESMF_GridComp) :: gridcomp2
```

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:

Assign gridcomp1 as an alias to the same ESMF GridComp object in memory as gridcomp2. If gridcomp2 is invalid, then gridcomp1 will be equally invalid after the assignment.

The arguments are:

gridcomp1 The `ESMF_GridComp` object on the left hand side of the assignment.

gridcomp2 The `ESMF_GridComp` object on the right hand side of the assignment.

17.4.2 ESMF_GridCompOperator(==) - GridComp equality operator

INTERFACE:

```
interface operator(==)
  if (gridcomp1 == gridcomp2) then ... endif
            OR
  result = (gridcomp1 == gridcomp2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_GridComp), intent(in) :: gridcomp1
type(ESMF_GridComp), intent(in) :: gridcomp2
```

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:

Test whether gridcomp1 and gridcomp2 are valid aliases to the same ESMF GridComp object in memory. For a more general comparison of two ESMF GridComps, going beyond the simple alias test, the ESMF_GridCompMatch() function (not yet implemented) must be used.

The arguments are:

gridcomp1 The `ESMF_GridComp` object on the left hand side of the equality operation.

gridcomp2 The `ESMF_GridComp` object on the right hand side of the equality operation.

17.4.3 ESMF_GridCompOperator(/=) - GridComp not equal operator

INTERFACE:

```
interface operator(/=)
  if (gridcomp1 /= gridcomp2) then ... endif
            OR
  result = (gridcomp1 /= gridcomp2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_GridComp), intent(in) :: gridcomp1
type(ESMF_GridComp), intent(in) :: gridcomp2
```

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:

Test whether gridcomp1 and gridcomp2 are *not* valid aliases to the same ESMF GridComp object in memory. For a more general comparison of two ESMF GridComps, going beyond the simple alias test, the ESMF_GridCompMatch() function (not yet implemented) must be used.

The arguments are:

gridcomp1 The ESMF_GridComp object on the left hand side of the non-equality operation.

gridcomp2 The ESMF_GridComp object on the right hand side of the non-equality operation.

17.4.4 ESMF_GridCompCreate - Create a GridComp

INTERFACE:

```
recursive function ESMF_GridCompCreate(grid, gridList, &
  mesh, meshList, locstream, locstreamList, xgrid, xgridList, &
  config, configFile, clock, petList, contextflag, name, rc)
```

RETURN VALUE:

```
type(ESMF_GridComp) :: ESMF_GridCompCreate
```

ARGUMENTS:

```
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_Grid),         intent(in),    optional :: grid
  type(ESMF_Grid),         intent(in),    optional :: gridList(:)
  type(ESMF_Mesh),         intent(in),    optional :: mesh
  type(ESMF_Mesh),         intent(in),    optional :: meshList(:)
  type(ESMF_LocStream),    intent(in),    optional :: locstream
  type(ESMF_LocStream),    intent(in),    optional :: locstreamList(:)
  type(ESMF_XGrid),        intent(in),    optional :: xgrid
  type(ESMF_XGrid),        intent(in),    optional :: xgridList(:)
  type(ESMF_Config),       intent(in),    optional :: config
  character(len=*),        intent(in),    optional :: configFile
  type(ESMF_Clock),        intent(in),    optional :: clock
  integer,                 intent(in),    optional :: petList(:)
  type(ESMF_Context_Flag), intent(in),    optional :: contextflag
  character(len=*),        intent(in),    optional :: name
  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:

  7.1.0r Added arguments gridList, mesh, meshList, locstream, locstreamList, xgrid, and xgridList. These arguments add support for holding references to multiple geom objects, either of the same type, or different type, in the same ESMF_GridComp object.

DESCRIPTION:

This interface creates an ESMF_GridComp object. By default, a separate VM context will be created for each component. This implies creating a new MPI communicator and allocating additional memory to manage the VM resources. When running on a large number of processors, creating a separate VM for each component could be both time and memory inefficient. If the application is sequential, i.e., each component is running on all the PETs of the global VM, it will be more efficient to use the global VM instead of creating a new one. This can be done by setting contextflag to ESMF_CONTEXT_PARENT_VM.

The return value is the new ESMF_GridComp.

The arguments are:

[grid] Associate an ESMF_Grid object with the newly created component. This is simply a convenience feature for the user. The ESMF library code does not access the grid object. The grid argument is mutually exclusive with the gridList argument. If both arguments are provided, the routine will fail, and an error is returned in rc. By default, i.e. if neither grid nor gridList are provided, no ESMF_Grid objects are associated with the component.

[gridList] Associate a list of ESMF_Grid objects with the newly created component. This is simply a convenience feature for the user. The ESMF library code does not access the gridList object. The gridList argument is mutually exclusive with the grid argument. If both arguments are provided, the routine will fail, and an error is returned in rc. By default, i.e. if neither grid nor gridList are provided, no ESMF_Grid objects are associated with the component.

[mesh] Associate an ESMF_Mesh object with the newly created component. This is simply a convenience feature for the user. The ESMF library code does not access the mesh object. The mesh argument is mutually exclusive with the meshList argument. If both arguments are provided, the routine will fail, and an error is returned in rc. By default, i.e. if neither mesh nor meshList are provided, no ESMF_Mesh objects are associated with the component.

[meshList] Associate a list of ESMF_Mesh objects with the newly created component. This is simply a convenience feature for the user. The ESMF library code does not access the meshList object. The meshList argument is mutually exclusive with the mesh argument. If both arguments are provided, the routine will fail, and an error is returned in rc. By default, i.e. if neither mesh nor meshList are provided, no ESMF_Mesh objects are associated with the component.

[locstream] Associate an ESMF_LocStream object with the newly created component. This is simply a convenience feature for the user. The ESMF library code does not access the locstream object. The locstream argument is mutually exclusive with the locstreamList argument. If both arguments are provided, the routine will fail, and an error is returned in rc. By default, i.e. if neither locstream nor locstreamList are provided, no ESMF_LocStream objects are associated with the component.

[locstreamList] Associate a list of ESMF_LocStream objects with the newly created component. This is simply a convenience feature for the user. The ESMF library code does not access the locstreamList object. The locstreamList argument is mutually exclusive with the locstream argument. If both arguments are provided, the routine will fail, and an error is returned in rc. By default, i.e. if neither locstream nor locstreamList are provided, no ESMF_LocStream objects are associated with the component.

[xgrid] Associate an ESMF_XGrid object with the newly created component. This is simply a convenience feature for the user. The ESMF library code does not access the xgrid object. The xgrid argument is mutually exclusive with the xgridList argument. If both arguments are provided, the routine will fail, and an error is returned in rc. By default, i.e. if neither xgrid nor xgridList are provided, no ESMF_XGrid objects are associated with the component.

[xgridList] Associate a list of ESMF_XGrid objects with the newly created component. This is simply a convenience feature for the user. The ESMF library code does not access the xgridList object. The xgridList argument is mutually exclusive with the xgrid argument. If both arguments are provided, the routine will fail, and

an error is returned in rc. By default, i.e. if neither xgrid nor xgridList are provided, no ESMF_XGrid objects are associated with the component.

[config] An already-created ESMF_Config object to be attached to the newly created component. If both config and configFile arguments are specified, config takes priority.

[configFile] The filename of an ESMF_Config format file. If specified, a new ESMF_Config object is created and attached to the newly created component. The configFile file is opened and associated with the new config object. If both config and configFile arguments are specified, config takes priority.

[clock] Component-specific ESMF_Clock. This clock is available to be queried and updated by the new ESMF_GridComp as it chooses. This should not be the parent component clock, which should be maintained and passed down to the initialize/run/finalize routines separately.

[petList] List of parent PETs given to the created child component by the parent component. If petList is not specified all of the parent PETs will be given to the child component. The order of PETs in petList determines how the child local PETs refer back to the parent PETs.

[contextflag] Specify the component's VM context. The default context is ESMF_CONTEXT_OWN_VM. See section **??** for a complete list of valid flags.

[name] Name of the newly-created ESMF_GridComp. This name can be altered from within the ESMF_GridComp code once the initialization routine is called.

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

17.4.5 ESMF_GridCompDestroy - Release resources associated with a GridComp

INTERFACE:

```
recursive subroutine ESMF_GridCompDestroy(gridcomp, &
  timeout, timeoutFlag, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp), intent(inout)          :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  integer,             intent(in),   optional :: timeout
  logical,             intent(out),  optional :: timeoutFlag
  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:

5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Destroys an `ESMF_GridComp`, releasing the resources associated with the object.

The arguments are:

gridcomp Release all resources associated with this `ESMF_GridComp` and mark the object as invalid. It is an error to pass this object into any other routines after being destroyed.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The `timeout` argument is only supported for connected dual components.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

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

17.4.6 ESMF_GridCompFinalize - Call the GridComp's finalize routine

INTERFACE:

```
recursive subroutine ESMF_GridCompFinalize(gridcomp, &
  importState, exportState, clock, syncflag, phase, timeout, timeoutFlag, &
  userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp),  intent(inout)           :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_State),     intent(inout), optional :: importState
  type(ESMF_State),     intent(inout), optional :: exportState
  type(ESMF_Clock),     intent(inout), optional :: clock
  type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
  integer,              intent(in),    optional :: phase
  integer,              intent(in),    optional :: timeout
  logical,              intent(out),   optional :: timeoutFlag
  integer,              intent(out),   optional :: userRc
  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:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Call the associated user-supplied finalization routine for an `ESMF_GridComp`.

The arguments are:

gridcomp The `ESMF_GridComp` to call finalize routine for.

[importState] `ESMF_State` containing import data. If not present, a dummy argument will be passed to the user-supplied routine. The importState argument in the user code cannot be optional.

[exportState] `ESMF_State` containing export data. If not present, a dummy argument will be passed to the user-supplied routine. The exportState argument in the user code cannot be optional.

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

[syncflag] Blocking behavior of this method call. See section **??** for a list of valid blocking options. Default option is `ESMF_SYNC_VASBLOCKING` which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.

[phase] Component providers must document whether each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accommodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument is optional. The default is 1.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The `timeout` argument is only supported for connected dual components.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

[userRc] Return code set by `userRoutine` before returning.

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

17.4.7 ESMF_GridCompGet - Get GridComp information

INTERFACE:

```
  recursive subroutine ESMF_GridCompGet(gridcomp, &
    gridIsPresent, grid, gridList, meshIsPresent, mesh, meshList, &
    locstreamIsPresent, locstream, locstreamList, xgridIsPresent, &
    xgrid, xgridList, importStateIsPresent, importState, &
    exportStateIsPresent, exportState, configIsPresent, config, &
    configFileIsPresent, configFile, clockIsPresent, clock, localPet, &
    petCount, contextflag, currentMethod, currentPhase, comptype, &
    vmIsPresent, vm, name, rc)
```

ARGUMENTS:

```
    type(ESMF_GridComp),           intent(in)            :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    logical,                       intent(out), optional :: gridIsPresent
    type(ESMF_Grid),               intent(out), optional :: grid
    type(ESMF_Grid), allocatable,  intent(out), optional :: gridList(:)
    logical,                       intent(out), optional :: meshIsPresent
    type(ESMF_Mesh),               intent(out), optional :: mesh
    type(ESMF_Mesh), allocatable,  intent(out), optional :: meshList(:)
    logical,                       intent(out), optional :: locstreamIsPresent
    type(ESMF_LocStream),          intent(out), optional :: locstream
    type(ESMF_LocStream), allocatable, intent(out), optional :: locstreamList(:)
    logical,                       intent(out), optional :: xgridIsPresent
    type(ESMF_XGrid),              intent(out), optional :: xgrid
    type(ESMF_XGrid), allocatable, intent(out), optional :: xgridList(:)
    logical,                       intent(out), optional :: importStateIsPresent
    type(ESMF_State),              intent(out), optional :: importState
    logical,                       intent(out), optional :: exportStateIsPresent
    type(ESMF_State),              intent(out), optional :: exportState
    logical,                       intent(out), optional :: configIsPresent
    type(ESMF_Config),             intent(out), optional :: config
    logical,                       intent(out), optional :: configFileIsPresent
    character(len=*),              intent(out), optional :: configFile
    logical,                       intent(out), optional :: clockIsPresent
    type(ESMF_Clock),              intent(out), optional :: clock
    integer,                       intent(out), optional :: localPet
    integer,                       intent(out), optional :: petCount
    type(ESMF_Context_Flag),       intent(out), optional :: contextflag
    type(ESMF_Method_Flag),        intent(out), optional :: currentMethod
    integer,                       intent(out), optional :: currentPhase
    type(ESMF_CompType_Flag),      intent(out), optional :: comptype
    logical,                       intent(out), optional :: vmIsPresent
    type(ESMF_VM),                 intent(out), optional :: vm
    character(len=*),              intent(out), optional :: name
    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:

  7.1.0r Added arguments `gridList`, `meshIsPresent`, `mesh`, `meshList`, `locstreamIsPresent`, `locstream`, `locstreamList`, `xgridIsPresent`, `xgrid`, and `xgridList`. These arguments add support for accessing references to multiple geom objects, either of the same type, or different type, associated with the same `ESMF_GridComp` object.

DESCRIPTION:

Get information about an `ESMF_GridComp` object.

The arguments are:

gridcomp The `ESMF_GridComp` object being queried.

[gridIsPresent] Set to `.true.` if at least one `ESMF_Grid` object is associated with the `gridcomp` component. Set to `.false.` otherwise.

[grid] Return the `ESMF_Grid` object associated with the `gridcomp` component. If multiple `ESMF_Grid` objects are associated, return the first in the list. It is an error to query for `grid` if no `ESMF_Grid` object is associated with the `gridcomp` component. If unsure, query for `gridIsPresent` first, or use the `gridList` variant.

[gridList] Return a list of all `ESMF_Grid` objects associated with the `gridcomp` component. The size of the returned `gridList` corresponds to the number of `ESMF_Grid` objects associated. If no `ESMF_Grid` object is associated with the `gridcomp` component, the size of the returned `gridList` is zero.

[meshIsPresent] Set to `.true.` if at least one `ESMF_Mesh` object is associated with the `gridcomp` component. Set to `.false.` otherwise.

[mesh] Return the `ESMF_Mesh` object associated with the `gridcomp` component. If multiple `ESMF_Mesh` objects are associated, return the first in the list. It is an error to query for `mesh` if no `ESMF_Mesh` object is associated with the `gridcomp` component. If unsure, query for `meshIsPresent` first, or use the `meshList` variant.

[meshList] Return a list of all `ESMF_Mesh` objects associated with the `gridcomp` component. The size of the returned `meshList` corresponds to the number of `ESMF_Mesh` objects associated. If no `ESMF_Mesh` object is associated with the `gridcomp` component, the size of the returned `meshList` is zero.

[locstreamIsPresent] Set to `.true.` if at least one `ESMF_LocStream` object is associated with the `gridcomp` component. Set to `.false.` otherwise.

[locstream] Return the `ESMF_LocStream` object associated with the `gridcomp` component. If multiple `ESMF_LocStream` objects are associated, return the first in the list. It is an error to query for `locstream` if no `ESMF_Grid` object is associated with the `gridcomp` component. If unsure, query for `locstreamIsPresent` first, or use the `locstreamList` variant.

[locstreamList] Return a list of all `ESMF_LocStream` objects associated with the `gridcomp` component. The size of the returned `locstreamList` corresponds to the number of `ESMF_LocStream` objects associated. If no `ESMF_LocStream` object is associated with the `gridcomp` component, the size of the returned `locstreamList` is zero.

[xgridIsPresent] Set to `.true.` if at least one `ESMF_XGrid` object is associated with the `gridcomp` component. Set to `.false.` otherwise.

[xgrid] Return the ESMF_XGrid object associated with the gridcomp component. If multiple ESMF_XGrid objects are associated, return the first in the list. It is an error to query for xgrid if no ESMF_XGrid object is associated with the gridcomp component. If unsure, query for xgridIsPresent first, or use the xgridList variant.

[xgridList] Return a list of all ESMF_XGrid objects associated with the gridcomp component. The size of the returned xgridList corresponds to the number of ESMF_XGrid objects associated. If no ESMF_XGrid object is associated with the gridcomp component, the size of the returned xgridList is zero.

[importStateIsPresent] .true. if importState was set in GridComp object, .false. otherwise.

[importState] Return the associated import State. It is an error to query for the import State if none is associated with the GridComp. If unsure, get importStateIsPresent first to determine the status.

[exportStateIsPresent] .true. if exportState was set in GridComp object, .false. otherwise.

[exportState] Return the associated export State. It is an error to query for the export State if none is associated with the GridComp. If unsure, get exportStateIsPresent first to determine the status.

[configIsPresent] .true. if config was set in GridComp object, .false. otherwise.

[config] Return the associated Config. It is an error to query for the Config if none is associated with the GridComp. If unsure, get configIsPresent first to determine the status.

[configFileIsPresent] .true. if configFile was set in GridComp object, .false. otherwise.

[configFile] Return the associated configuration filename. It is an error to query for the configuration filename if none is associated with the GridComp. If unsure, get configFileIsPresent first to determine the status.

[clockIsPresent] .true. if clock was set in GridComp object, .false. otherwise.

[clock] Return the associated Clock. It is an error to query for the Clock if none is associated with the GridComp. If unsure, get clockIsPresent first to determine the status.

[localPet] Return the local PET id within the ESMF_GridComp object.

[petCount] Return the number of PETs in the the ESMF_GridComp object.

[contextflag] Return the ESMF_Context_Flag for this ESMF_GridComp. See section **??** for a complete list of valid flags.

[currentMethod] Return the current ESMF_Method_Flag of the ESMF_GridComp execution. See section **??** for a complete list of valid options.

[currentPhase] Return the current phase of the ESMF_GridComp execution.

[comptype] Return the Component type. See section **??** for a complete list of valid flags.

[vmIsPresent] .true. if vm was set in GridComp object, .false. otherwise.

[vm] Return the associated VM. It is an error to query for the VM if none is associated with the GridComp. If unsure, get vmIsPresent first to determine the status.

[name] Return the name of the ESMF_GridComp.

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

17.4.8 ESMF_GridCompGetInternalState - Get private data block pointer

INTERFACE:

```
subroutine ESMF_GridCompGetInternalState(gridcomp, wrappedDataPointer, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)             :: gridcomp
type(wrapper)                   :: wrappedDataPointer
integer,            intent(out) :: 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:

Available to be called by an `ESMF_GridComp` at any time after `ESMF_GridCompSetInternalState` has been called. Since init, run, and finalize must be separate subroutines, data that they need to share in common can either be module global data, or can be allocated in a private data block and the address of that block can be registered with the framework and retrieved by this call. When running multiple instantiations of an `ESMF_GridComp`, for example during ensemble runs, it may be simpler to maintain private data specific to each run with private data blocks. A corresponding `ESMF_GridCompSetInternalState` call sets the data pointer to this block, and this call retrieves the data pointer. Note that the `wrappedDataPointer` argument needs to be a derived type which contains only a pointer of the type of the data block defined by the user. When making this call the pointer needs to be unassociated. When the call returns, the pointer will now reference the original data block which was set during the previous call to `ESMF_GridCompSetInternalState`.

Only the *last* data block set via `ESMF_GridCompSetInternalState` will be accessible.

CAUTION: If you are working with a compiler that does not support Fortran 2018 assumed-type dummy arguments, then this method does not have an explicit Fortran interface. In this case do not specify argument keywords when calling this method!

The arguments are:

gridcomp An `ESMF_GridComp` object.

wrappedDataPointer A derived type (wrapper), containing only an unassociated pointer to the private data block. The framework will fill in the pointer. When this call returns, the pointer is set to the same address set during the last `ESMF_GridCompSetInternalState` call. This level of indirection is needed to reliably set and retrieve the data block no matter which architecture or compiler is used.

rc Return code; equals `ESMF_SUCCESS` if there are no errors. Note: unlike most other ESMF routines, this argument is not optional because of implementation considerations.

17.4.9 ESMF_GridCompInitialize - Call the GridComp's initialize routine

INTERFACE:

```
recursive subroutine ESMF_GridCompInitialize(gridcomp, &
  importState, exportState, clock, syncflag, phase, timeout, timeoutFlag, &
  userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp),  intent(inout)           :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_State),     intent(inout), optional :: importState
  type(ESMF_State),     intent(inout), optional :: exportState
  type(ESMF_Clock),     intent(inout), optional :: clock
  type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
  integer,              intent(in),    optional :: phase
  integer,              intent(in),    optional :: timeout
  logical,              intent(out),   optional :: timeoutFlag
  integer,              intent(out),   optional :: userRc
  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:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Call the associated user initialization routine for an `ESMF_GridComp`.

The arguments are:

gridcomp `ESMF_GridComp` to call initialize routine for.

[importState] `ESMF_State` containing import data for coupling. If not present, a dummy argument will be passed to the user-supplied routine. The importState argument in the user code cannot be optional.

[exportState] `ESMF_State` containing export data for coupling. If not present, a dummy argument will be passed to the user-supplied routine. The exportState argument in the user code cannot be optional.

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

[syncflag] Blocking behavior of this method call. See section **??** for a list of valid blocking options. Default option is `ESMF_SYNC_VASBLOCKING` which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.

[phase] Component providers must document whether each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accommodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument is optional. The default is 1.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The `timeout` argument is only supported for connected dual components.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

[userRc] Return code set by `userRoutine` before returning.

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

17.4.10 ESMF_GridCompIsCreated - Check whether a GridComp object has been created

INTERFACE:

```
function ESMF_GridCompIsCreated(gridcomp, rc)
```

RETURN VALUE:

```
logical :: ESMF_GridCompIsCreated
```

ARGUMENTS:

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

DESCRIPTION:

Return `.true.` if the `gridcomp` has been created. Otherwise return `.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:

gridcomp `ESMF_GridComp` queried.

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

17.4.11 ESMF_GridCompIsPetLocal - Inquire if this GridComp is to execute on the calling PET

INTERFACE:

```
recursive function ESMF_GridCompIsPetLocal(gridcomp, rc)
```

RETURN VALUE:

```
logical :: ESMF_GridCompIsPetLocal
```

ARGUMENTS:

```
type(ESMF_GridComp), intent(in)            :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
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:

Inquire if this `ESMF_GridComp` object is to execute on the calling PET.

The return value is `.true.` if the component is to execute on the calling PET, `.false.` otherwise.

The arguments are:

gridcomp `ESMF_GridComp` queried.

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

17.4.12 ESMF_GridCompPrint - Print GridComp information

INTERFACE:

```
subroutine ESMF_GridCompPrint(gridcomp, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp), intent(in)            :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
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:

Prints information about an `ESMF_GridComp` to `stdout`.

The arguments are:

gridcomp `ESMF_GridComp` to print.

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

17.4.13 ESMF_GridCompReadRestart - Call the GridComp's read restart routine

INTERFACE:

```
recursive subroutine ESMF_GridCompReadRestart(gridcomp, &
  importState, exportState, clock, syncflag, phase, timeout, timeoutFlag, &
  userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp),  intent(inout)           :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_State),     intent(inout), optional :: importState
  type(ESMF_State),     intent(inout), optional :: exportState
  type(ESMF_Clock),     intent(inout), optional :: clock
  type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
  integer,              intent(in),    optional :: phase
  integer,              intent(in),    optional :: timeout
  logical,              intent(out),   optional :: timeoutFlag
  integer,              intent(out),   optional :: userRc
  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:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Call the associated user read restart routine for an ESMF_GridComp.

The arguments are:

gridcomp ESMF_GridComp to call run routine for.

[importState] ESMF_State containing import data. If not present, a dummy argument will be passed to the user-supplied routine. The importState argument in the user code cannot be optional.

[exportState] ESMF_State containing export data. If not present, a dummy argument will be passed to the user-supplied routine. The exportState argument in the user code cannot be optional.

[clock] External ESMF_Clock for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

[syncflag] Blocking behavior of this method call. See section **??** for a list of valid blocking options. Default option is ESMF_SYNC_VASBLOCKING which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.

[phase] Component providers must document whether each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accommodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument is optional. The default is 1.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The timeout argument is only supported for connected dual components.

[timeoutFlag] Returns .true. if the timeout was reached, .false. otherwise. If timeoutFlag was *not* provided, a timeout condition will lead to a return code of rc \= ESMF_SUCCESS. Otherwise the return value of timeoutFlag is the sole indicator of a timeout condition.

[userRc] Return code set by userRoutine before returning.

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

17.4.14 ESMF_GridCompRun - Call the GridComp's run routine

INTERFACE:

```
recursive subroutine ESMF_GridCompRun(gridcomp, &
  importState, exportState, clock, syncflag, phase, timeout, timeoutFlag, &
  userRc, rc)
```

ARGUMENTS:

```
    type(ESMF_GridComp),  intent(inout)           :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_State),     intent(inout), optional :: importState
    type(ESMF_State),     intent(inout), optional :: exportState
    type(ESMF_Clock),     intent(inout), optional :: clock
    type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
    integer,              intent(in),    optional :: phase
    integer,              intent(in),    optional :: timeout
    logical,              intent(out),   optional :: timeoutFlag
    integer,              intent(out),   optional :: userRc
    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:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Call the associated user run routine for an `ESMF_GridComp`.

The arguments are:

gridcomp `ESMF_GridComp` to call run routine for.

[importState] `ESMF_State` containing import data. If not present, a dummy argument will be passed to the user-supplied routine. The importState argument in the user code cannot be optional.

[exportState] `ESMF_State` containing export data. If not present, a dummy argument will be passed to the user-supplied routine. The exportState argument in the user code cannot be optional.

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

[syncflag] Blocking behavior of this method call. See section **??** for a list of valid blocking options. Default option is `ESMF_SYNC_VASBLOCKING` which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.

[phase] Component providers must document whether each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accommodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument is optional. The default is 1.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The `timeout` argument is only supported for connected dual components.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

[userRc] Return code set by `userRoutine` before returning.

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

17.4.15 ESMF_GridCompServiceLoop - Call the GridComp's service loop routine

INTERFACE:

```
   recursive subroutine ESMF_GridCompServiceLoop(gridcomp, &
     importState, exportState, clock, syncflag, port, timeout, timeoutFlag, rc)
```

ARGUMENTS:

```
     type(ESMF_GridComp),  intent(inout)           :: gridcomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_State),     intent(inout), optional :: importState
     type(ESMF_State),     intent(inout), optional :: exportState
     type(ESMF_Clock),     intent(inout), optional :: clock
     type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
     integer,              intent(in),    optional :: port
     integer,              intent(in),    optional :: timeout
     logical,              intent(out),   optional :: timeoutFlag
     integer,              intent(out),   optional :: rc
```

DESCRIPTION:

Call the ServiceLoop routine for an `ESMF_GridComp`. This tries to establish a "component tunnel" between the *actual* Component (calling this routine) and a `dual` Component connecting to it through a matching SetServices call.

The arguments are:

gridcomp `ESMF_GridComp` to call service loop routine for.

[importState] `ESMF_State` containing import data for coupling. If not present, a dummy argument will be passed to the user-supplied routine. The importState argument in the user code cannot be optional.

[exportState] `ESMF_State` containing export data for coupling. If not present, a dummy argument will be passed to the user-supplied routine. The exportState argument in the user code cannot be optional.

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

[syncflag] Blocking behavior of this method call. See section **??** for a list of valid blocking options. Default option is `ESMF_SYNC_VASBLOCKING` which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.

[port] In case a port number is provided, the "component tunnel" is established using sockets. The actual component side, i.e. the side that calls into `ESMF_GridCompServiceLoop()`, starts to listen on the specified port as the server. The valid port range is [1024, 65535]. In case the `port` argument is *not* specified, the "component tunnel" is established within the same executable using local communication methods (e.g. MPI).

[timeout] The maximum period in seconds that this call will wait for communications with the dual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. (NOTE: Currently this option is only available for socket based component tunnels.)

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

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

17.4.16 ESMF_GridCompSet - Set or reset information about the GridComp

INTERFACE:

```
  subroutine ESMF_GridCompSet(gridcomp, grid, gridList, &
    mesh, meshList, locstream, locstreamList, xgrid, xgridList, &
    config, configFile, clock, name, rc)
```

ARGUMENTS:

```
    type(ESMF_GridComp),    intent(inout)         :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_Grid),        intent(in),  optional :: grid
    type(ESMF_Grid),        intent(in),  optional :: gridList(:)
    type(ESMF_Mesh),        intent(in),  optional :: mesh
    type(ESMF_Mesh),        intent(in),  optional :: meshList(:)
    type(ESMF_LocStream),   intent(in),  optional :: locstream
    type(ESMF_LocStream),   intent(in),  optional :: locstreamList(:)
    type(ESMF_XGrid),       intent(in),  optional :: xgrid
    type(ESMF_XGrid),       intent(in),  optional :: xgridList(:)
    type(ESMF_Config),      intent(in),  optional :: config
    character(len=*),       intent(in),  optional :: configFile
    type(ESMF_Clock),       intent(in),  optional :: clock
    character(len=*),       intent(in),  optional :: name
    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:

  7.1.0r Added arguments `gridList`, `mesh`, `meshList`, `locstream`, `locstreamList`, `xgrid`, and `xgridList`. These arguments add support for holding references to multiple geom objects, either of the same type, or different type, in the same `ESMF_GridComp` object.

DESCRIPTION:

Sets or resets information about an `ESMF_GridComp`.

The arguments are:

gridcomp `ESMF_GridComp` to change.

[grid] Associate an `ESMF_Grid` object with the `gridcomp` component. This is simply a convenience feature for the user. The ESMF library code does not access the `grid` object. The `grid` argument is mutually exclusive with the `gridList` argument. If both arguments are provided, the routine will fail, and an error is returned in `rc`. By default, i.e. if neither `grid` nor `gridList` are provided, the `ESMF_Grid` association of the incoming `gridcomp` component remains unchanged.

[gridList] Associate a list of `ESMF_Grid` objects with the `gridcomp` component. This is simply a convenience feature for the user. The ESMF library code does not access the `gridList` object. The `gridList` argument is mutually exclusive with the `grid` argument. If both arguments are provided, the routine will fail, and an error is returned in `rc`. By default, i.e. if neither `grid` nor `gridList` are provided, the `ESMF_Grid` association of the incoming `gridcomp` component remains unchanged.

[mesh] Associate an `ESMF_Mesh` object with the `gridcomp` component. This is simply a convenience feature for the user. The ESMF library code does not access the `mesh` object. The `mesh` argument is mutually exclusive with the `meshList` argument. If both arguments are provided, the routine will fail, and an error is returned in `rc`. By default, i.e. if neither `mesh` nor `meshList` are provided, the `ESMF_Mesh` association of the incoming `gridcomp` component remains unchanged.

[meshList] Associate a list of `ESMF_Mesh` objects with the `gridcomp` component. This is simply a convenience feature for the user. The ESMF library code does not access the `meshList` object. The `meshList` argument is mutually exclusive with the `mesh` argument. If both arguments are provided, the routine will fail, and an error is returned in `rc`. By default, i.e. if neither `mesh` nor `meshList` are provided, the `ESMF_Mesh` association of the incoming `gridcomp` component remains unchanged.

[locstream] Associate an `ESMF_LocStream` object with the `gridcomp` component. This is simply a convenience feature for the user. The ESMF library code does not access the `locstream` object. The `locstream` argument is mutually exclusive with the `locstreamList` argument. If both arguments are provided, the routine will fail, and an error is returned in `rc`. By default, i.e. if neither `locstream` nor `locstreamList` are provided, the `ESMF_LocStream` association of the incoming `gridcomp` component remains unchanged.

[locstreamList] Associate a list of `ESMF_LocStream` objects with the `gridcomp` component. This is simply a convenience feature for the user. The ESMF library code does not access the `locstreamList` object. The `locstreamList` argument is mutually exclusive with the `locstream` argument. If both arguments are provided, the routine will fail, and an error is returned in `rc`. By default, i.e. if neither `locstream` nor `locstreamList` are provided, the `ESMF_LocStream` association of the incoming `gridcomp` component remains unchanged.

[xgrid] Associate an `ESMF_XGrid` object with the `gridcomp` component. This is simply a convenience feature for the user. The ESMF library code does not access the `xgrid` object. The `xgrid` argument is mutually exclusive

with the xgridList argument. If both arguments are provided, the routine will fail, and an error is returned in rc. By default, i.e. if neither xgrid nor xgridList are provided, the ESMF_XGrid association of the incoming gridcomp component remains unchanged.

[xgridList] Associate a list of ESMF_XGrid objects with the gridcomp component. This is simply a convenience feature for the user. The ESMF library code does not access the xgridList object. The xgridList argument is mutually exclusive with the xgrid argument. If both arguments are provided, the routine will fail, and an error is returned in rc. By default, i.e. if neither xgrid nor xgridList are provided, the ESMF_XGrid association of the incoming gridcomp component remains unchanged.

[config] An already-created ESMF_Config object to be attached to the component. If both config and configFile arguments are specified, config takes priority.

[configFile] The filename of an ESMF_Config format file. If specified, a new ESMF_Config object is created and attached to the component. The configFile file is opened and associated with the new config object. If both config and configFile arguments are specified, config takes priority.

[clock] Set the private clock for this ESMF_GridComp.

[name] Set the name of the ESMF_GridComp.

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

17.4.17 ESMF_GridCompSetEntryPoint - Set user routine as entry point for standard GridComp method

INTERFACE:

```
   recursive subroutine ESMF_GridCompSetEntryPoint(gridcomp, methodflag, &
     userRoutine, phase, rc)
```

ARGUMENTS:

```
     type(ESMF_GridComp),    intent(inout)         :: gridcomp
     type(ESMF_Method_Flag), intent(in)            :: methodflag
     interface
       subroutine userRoutine(gridcomp, importState, exportState, clock, rc)
         use ESMF_CompMod
         use ESMF_StateMod
         use ESMF_ClockMod
         implicit none
         type(ESMF_GridComp)         :: gridcomp     ! 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
       end subroutine
     end interface
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                intent(in),  optional :: phase
     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:

Registers a user-supplied `userRoutine` as the entry point for one of the predefined Component `methodflag`s. After this call the `userRoutine` becomes accessible via the standard Component method API.

The arguments are:

gridcomp An `ESMF_GridComp` object.

methodflag One of a set of predefined Component methods - e.g. `ESMF_METHOD_INITIALIZE`, `ESMF_METHOD_RUN`, `ESMF_METHOD_FINALIZE`. See section **??** for a complete list of valid method options.

userRoutine The user-supplied subroutine to be associated for this Component `method`. Argument types, intent and order must match the interface signature, and must not have the `optional` attribute. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

[phase] The `phase` number for multi-phase methods. For single phase methods the `phase` argument can be omitted. The default setting is 1.

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

17.4.18 ESMF_GridCompSetInternalState - Set private data block pointer

INTERFACE:

```
subroutine ESMF_GridCompSetInternalState(gridcomp, wrappedDataPointer, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)             :: gridcomp
type(wrapper)                   :: wrappedDataPointer
integer,            intent(out) :: 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:

Available to be called by an ESMF_GridComp at any time, but expected to be most useful when called during the registration process, or initialization. Since init, run, and finalize must be separate subroutines, data that they need to share in common can either be module global data, or can be allocated in a private data block and the address of that block can be registered with the framework and retrieved by subsequent calls. When running multiple instantiations of an ESMF_GridComp, for example during ensemble runs, it may be simpler to maintain private data specific to each run with private data blocks. A corresponding ESMF_GridCompGetInternalState call retrieves the data pointer.

Only the *last* data block set via ESMF_GridCompSetInternalState will be accessible.

CAUTION: If you are working with a compiler that does not support Fortran 2018 assumed-type dummy arguments, then this method does not have an explicit Fortran interface. In this case do not specify argument keywords when calling this method!

The arguments are:

gridcomp An ESMF_GridComp object.

wrappedDataPointer A pointer to the private data block, wrapped in a derived type which contains only a pointer to the block. This level of indirection is needed to reliably set and retrieve the data block no matter which architecture or compiler is used.

rc Return code; equals ESMF_SUCCESS if there are no errors. Note: unlike most other ESMF routines, this argument is not optional because of implementation considerations.

17.4.19 ESMF_GridCompSetServices - Call user routine to register GridComp methods

INTERFACE:

```
recursive subroutine ESMF_GridCompSetServices(gridcomp, &
  userRoutine, userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp), intent(inout)         :: gridcomp
  interface
    subroutine userRoutine(gridcomp, rc)
      use ESMF_CompMod
      implicit none
      type(ESMF_GridComp)        :: gridcomp ! must not be optional
      integer, intent(out)       :: rc       ! must not be optional
    end subroutine
  end interface
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  integer,             intent(out), optional :: userRc
  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:

Call into user provided `userRoutine` which is responsible for setting Component's Initialize(), Run(), and Finalize() services.

The arguments are:

gridcomp Gridded Component.

userRoutine The Component writer must supply a subroutine with the exact interface shown above for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

The `userRoutine`, when called by the framework, must make successive calls to `ESMF_GridCompSetEntryPoint()` to preset callback routines for standard Component Initialize(), Run(), and Finalize() methods.

[userRc] Return code set by `userRoutine` before returning.

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

17.4.20 ESMF_GridCompSetServices - Call user routine through name lookup, to register GridComp methods

INTERFACE:

```
! Private name; call using ESMF_GridCompSetServices()
recursive subroutine ESMF_GridCompSetServicesShObj(gridcomp, userRoutine, &
  sharedObj, userRoutineFound, userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp), intent(inout)          :: gridcomp
  character(len=*),    intent(in)             :: userRoutine
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  character(len=*),    intent(in),  optional :: sharedObj
  logical,             intent(out), optional :: userRoutineFound
  integer,             intent(out), optional :: userRc
  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:

  6.3.0r Added argument `userRoutineFound`. The new argument provides a way to test availability without causing error conditions.

DESCRIPTION:

Call into a user provided routine which is responsible for setting Component's Initialize(), Run(), and Finalize() services. The named `userRoutine` must exist in the executable, or in the shared object specified by `sharedObj`. In the latter case all of the platform specific details about dynamic linking and loading apply.

The arguments are:

gridcomp Gridded Component.

userRoutine Name of routine to be called, specified as a character string. The Component writer must supply a subroutine with the exact interface shown for `userRoutine` below. Arguments must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

INTERFACE:

```
interface
  subroutine userRoutine(gridcomp, rc)
    type(ESMF_GridComp)  :: gridcomp   ! must not be optional
    integer, intent(out) :: rc         ! must not be optional
  end subroutine
end interface
```

DESCRIPTION:

The `userRoutine`, when called by the framework, must make successive calls to `ESMF_GridCompSetEntryPoint()` to preset callback routines for standard Component Initialize(), Run(), and Finalize() methods.

[sharedObj] Name of shared object that contains `userRoutine`. If the `sharedObj` argument is not provided the executable itself will be searched for `userRoutine`.

[userRoutineFound] Report back whether the specified `userRoutine` was found and executed, or was not available. If this argument is present, not finding the `userRoutine` will not result in returning an error in `rc`. The default is to return an error if the `userRoutine` cannot be found.

[userRc] Return code set by `userRoutine` before returning.

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

17.4.21 ESMF_GridCompSetServices - Set to serve as Dual Component for an Actual Component

INTERFACE:

```
   ! Private name; call using ESMF_GridCompSetServices()
   recursive subroutine ESMF_GridCompSetServicesComp(gridcomp, &
     actualGridcomp, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Set the services of a Gridded Component to serve a "dual" Component for an "actual" Component. The component tunnel is VM based.

The arguments are:

gridcomp Dual Gridded Component.

actualGridcomp Actual Gridded Component.

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

17.4.22 ESMF_GridCompSetServices - Set to serve as Dual Component for an Actual Component through sockets

INTERFACE:

```
   ! Private name; call using ESMF_GridCompSetServices()
   recursive subroutine ESMF_GridCompSetServicesSock(gridcomp, port, &
     server, timeout, timeoutFlag, rc)
```

ARGUMENTS:

```
     type(ESMF_GridComp), intent(inout)         :: gridcomp
     integer,             intent(in)            :: port
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     character(len=*),    intent(in),  optional :: server
     integer,             intent(in),  optional :: timeout
     logical,             intent(out), optional :: timeoutFlag
     integer,             intent(out), optional :: rc
```

DESCRIPTION:

Set the services of a Gridded Component to serve a "dual" Component for an "actual" Component. The component tunnel is socket based.

The arguments are:

gridcomp Dual Gridded Component.

port Port number under which the actual component is being served. The valid port range is [1024, 65535].

[server] Server name where the actual component is being served. The default, i.e. if the `server` argument was not provided, is `localhost`.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

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

17.4.23 ESMF_GridCompSetVM - Call user routine to set GridComp VM properties

INTERFACE:

```
recursive subroutine ESMF_GridCompSetVM(gridcomp, userRoutine, &
  userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp), intent(inout)         :: gridcomp
  interface
    subroutine userRoutine(gridcomp, rc)
      use ESMF_CompMod
      implicit none
      type(ESMF_GridComp)        :: gridcomp ! must not be optional
      integer, intent(out)       :: rc       ! must not be optional
    end subroutine
  end interface
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  integer,             intent(out), optional :: userRc
  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:

Optionally call into user provided `userRoutine` which is responsible for setting Component's VM properties.

The arguments are:

gridcomp Gridded Component.

userRoutine The Component writer must supply a subroutine with the exact interface shown above for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

The subroutine, when called by the framework, is expected to use any of the `ESMF_GridCompSetVMxxx()` methods to set the properties of the VM associated with the Gridded Component.

[userRc] Return code set by `userRoutine` before returning.

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

17.4.24 ESMF_GridCompSetVM - Call user routine through name lookup, to set GridComp VM properties

INTERFACE:

```
! Private name; call using ESMF_GridCompSetVM()
recursive subroutine ESMF_GridCompSetVMShObj(gridcomp, userRoutine, &
  sharedObj, userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp), intent(inout)          :: gridcomp
  character(len=*),    intent(in)             :: userRoutine
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  character(len=*),    intent(in),  optional :: sharedObj
  integer,             intent(out), optional :: userRc
  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:

Optionally call into user provided `userRoutine` which is responsible for setting Component's VM properties. The named `userRoutine` must exist in the executable, or in the shared object specified by `sharedObj`. In the latter case all of the platform specific details about dynamic linking and loading apply.

The arguments are:

gridcomp Gridded Component.

userRoutine Routine to be called, specified as a character string. The Component writer must supply a subroutine with the exact interface shown for `userRoutine` below. Arguments must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

INTERFACE:

```
interface
  subroutine userRoutine(gridcomp, rc)
    type(ESMF_GridComp)  :: gridcomp    ! must not be optional
    integer, intent(out) :: rc          ! must not be optional
  end subroutine
end interface
```

DESCRIPTION:

The subroutine, when called by the framework, is expected to use any of the `ESMF_GridCompSetVMxxx()` methods to set the properties of the VM associated with the Gridded Component.

[sharedObj] Name of shared object that contains `userRoutine`. If the `sharedObj` argument is not provided the executable itself will be searched for `userRoutine`.

[userRc] Return code set by `userRoutine` before returning.

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

17.4.25 ESMF_GridCompSetVMMaxPEs - Associate PEs with PETs in GridComp VM

INTERFACE:

```
subroutine ESMF_GridCompSetVMMaxPEs(gridcomp, &
  maxPeCountPerPet, prefIntraProcess, prefIntraSsi, prefInterSsi, &
  pthreadMinStackSize, openMpHandling, openMpNumThreads, &
  forceChildPthreads, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp), intent(inout)         :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  integer,             intent(in),  optional :: maxPeCountPerPet
  integer,             intent(in),  optional :: prefIntraProcess
  integer,             intent(in),  optional :: prefIntraSsi
  integer,             intent(in),  optional :: prefInterSsi
  integer,             intent(in),  optional :: pthreadMinStackSize
  character(*),        intent(in),  optional :: openMpHandling
  integer,             intent(in),  optional :: openMpNumThreads
  logical,             intent(in),  optional :: forceChildPthreads
  integer,             intent(out), optional :: rc
```

DESCRIPTION:

Set characteristics of the ESMF_VM for this ESMF_GridComp. Attempts to associate up to maxPeCountPerPet PEs with each PET. Only PEs that are located on the same single system image (SSI) can be associated with the same PET. Within this constraint the call tries to get as close as possible to the number specified by maxPeCountPerPet.

The other constraint to this call is that the number of PEs is preserved. This means that the child Component in the end is associated with as many PEs as the parent Component provided to the child. The number of child PETs however is adjusted according to the above rule.

The typical use of ESMF_GridCompSetVMMaxPEs() is to allocate multiple PEs per PET in a Component for user-level threading, e.g. OpenMP.

The arguments are:

gridcomp ESMF_GridComp to set the ESMF_VM for.

[maxPeCountPerPet] Maximum number of PEs on each PET. Default for each SSI is the local number of PEs.

[prefIntraProcess] Communication preference within a single process. *Currently options not documented. Use default.*

[prefIntraSsi] Communication preference within a single system image (SSI). *Currently options not documented. Use default.*

[prefInterSsi] Communication preference between different single system images (SSIs). *Currently options not documented. Use default.*

[pthreadMinStackSize] Minimum stack size in byte of any child PET executing as Pthread. By default single threaded child PETs do *not* execute as Pthread, and their stack size is unaffected by this argument. However, for multi-threaded child PETs, or if forceChildPthreads is .true., child PETs execute as Pthreads with their own private stack.

For cases where OpenMP threads are used by the user code, each thread allocates its own private stack. For all threads *other* than the master, the stack size is set via the typical OMP_STACKSIZE environment variable mechanism. The PET itself, however, becomes the *master* of the OpenMP thread team, and is not affected by OMP_STACKSIZE. It is the master's stack that can be sized via the pthreadMinStackSize argument, and a large enough size is often critical.

When pthreadMinStackSize is absent, the default is to use the system default set by the limit or ulimit command. However, the stack of a Pthread cannot be unlimited, and a shell *stacksize* setting of *unlimited*, or any setting below the ESMF implemented minimum, will result in setting the stack size to 20MiB (the ESMF minimum). Depending on how much private data is used by the user code under the master thread, the default might be too small, and pthreadMinStackSize must be used to allocate sufficient stack space.

[openMpHandling] Handling of OpenMP threads. Supported options are:

- "none" - OpenMP handling is completely left to the user.
- "set" - ESMF uses the omp_set_num_threads() API to set the number of OpenMP threads in each team.
- "init" - ESMF sets the number of OpenMP threads in each team, and triggers the instantiation of the team.
- "pin" (default) - ESMF sets the number of OpenMP threads in each team, triggers the instantiation of the team, and pins each OpenMP thread to the corresponding PE.

[openMpNumThreads] Number of OpenMP threads in each OpenMP thread team. This can be any positive number. By default, or if openMpNumThreads is negative, each PET sets the number of OpenMP threads to its local peCount.

[forceChildPthreads] For `.true.`, force each child PET to execute in its own Pthread. By default, `.false.`, single PETs spawned from a parent PET execute in the same thread (or MPI process) as the parent PET. Multiple child PETs spawned by the same parent PET always execute as their own Pthreads.

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

17.4.26 ESMF_GridCompSetVMMaxThreads - Set multi-threaded PETs in GridComp VM

INTERFACE:

```
   subroutine ESMF_GridCompSetVMMaxThreads(gridcomp, &
     maxPetCountPerVas, prefIntraProcess, prefIntraSsi, prefInterSsi, &
     pthreadMinStackSize, forceChildPthreads, rc)
```

ARGUMENTS:

```
     type(ESMF_GridComp), intent(inout)         :: gridcomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,             intent(in),  optional :: maxPetCountPerVas
     integer,             intent(in),  optional :: prefIntraProcess
     integer,             intent(in),  optional :: prefIntraSsi
     integer,             intent(in),  optional :: prefInterSsi
     integer,             intent(in),  optional :: pthreadMinStackSize
     logical,             intent(in),  optional :: forceChildPthreads
     integer,             intent(out), optional :: rc
```

DESCRIPTION:

Set characteristics of the `ESMF_VM` for this `ESMF_GridComp`. Attempts to provide `maxPetCountPerVas` threaded PETs in each virtual address space (VAS). Only as many threaded PETs as there are PEs located on the single system image (SSI) can be associated with the VAS. Within this constraint the call tries to get as close as possible to the number specified by `maxPetCountPerVas`.

The other constraint to this call is that the number of PETs is preserved. This means that the child Component in the end is associated with as many PETs as the parent Component provided to the child. The threading level of the child PETs however is adjusted according to the above rule.

The typical use of `ESMF_GridCompSetVMMaxThreads()` is to run a Component multi-threaded with groups of PETs executing within a common virtual address space.

The arguments are:

gridcomp `ESMF_GridComp` to set the `ESMF_VM` for.

[maxPetCountPerVas] Maximum number of threaded PETs in each virtual address space (VAS). Default for each SSI is the local number of PEs.

[prefIntraProcess] Communication preference within a single process. *Currently options not documented. Use default.*

[prefIntraSsi] Communication preference within a single system image (SSI). *Currently options not documented. Use default.*

[prefInterSsi] Communication preference between different single system images (SSIs). *Currently options not documented. Use default.*

[pthreadMinStackSize] Minimum stack size in byte of any child PET executing as Pthread. By default single threaded child PETs do *not* execute as Pthread, and their stack size is unaffected by this argument. However, for multi-threaded child PETs, or if `forceChildPthreads` is `.true.`, child PETs execute as Pthreads with their own private stack.

For cases where OpenMP threads are used by the user code, each thread allocates its own private stack. For all threads *other* than the master, the stack size is set via the typical `OMP_STACKSIZE` environment variable mechanism. The PET itself, however, becomes the *master* of the OpenMP thread team, and is not affected by `OMP_STACKSIZE`. It is the master's stack that can be sized via the `pthreadMinStackSize` argument, and a large enough size is often critical.

When `pthreadMinStackSize` is absent, the default is to use the system default set by the `limit` or `ulimit` command. However, the stack of a Pthread cannot be unlimited, and a shell *stacksize* setting of *unlimited*, or any setting below the ESMF implemented minimum, will result in setting the stack size to 20MiB (the ESMF minimum). Depending on how much private data is used by the user code under the master thread, the default might be too small, and `pthreadMinStackSize` must be used to allocate sufficient stack space.

[forceChildPthreads] For `.true.`, force each child PET to execute in its own Pthread. By default, `.false.`, single PETs spawned from a parent PET execute in the same thread (or MPI process) as the parent PET. Multiple child PETs spawned by the same parent PET always execute as their own Pthreads.

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

17.4.27 ESMF_GridCompSetVMMinThreads - Set a reduced threading level in GridComp VM

INTERFACE:

```
  subroutine ESMF_GridCompSetVMMinThreads(gridcomp, &
    maxPeCountPerPet, prefIntraProcess, prefIntraSsi, prefInterSsi, &
    pthreadMinStackSize, forceChildPthreads, rc)
```

ARGUMENTS:

```
    type(ESMF_GridComp), intent(inout)         :: gridcomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,             intent(in),  optional :: maxPeCountPerPet
    integer,             intent(in),  optional :: prefIntraProcess
    integer,             intent(in),  optional :: prefIntraSsi
    integer,             intent(in),  optional :: prefInterSsi
    integer,             intent(in),  optional :: pthreadMinStackSize
    logical,             intent(in),  optional :: forceChildPthreads
    integer,             intent(out), optional :: rc
```

DESCRIPTION:

Set characteristics of the `ESMF_VM` for this `ESMF_GridComp`. Reduces the number of threaded PETs in each VAS. The `max` argument may be specified to limit the maximum number of PEs that a single PET can be associated with.

Several constraints apply: 1) the number of PEs cannot change, 2) PEs cannot migrate between single system images (SSIs), 3) the number of PETs cannot increase, only decrease, 4) PETs cannot migrate between virtual address spaces (VASs), nor can VASs migrate between SSIs.

The typical use of `ESMF_GridCompSetVMMinThreads()` is to run a Component across a set of single-threaded PETs.

The arguments are:

gridcomp `ESMF_GridComp` to set the `ESMF_VM` for.

[maxPeCountPerPet] Maximum number of PEs on each PET. Default for each SSI is the local number of PEs.

[prefIntraProcess] Communication preference within a single process. *Currently options not documented. Use default.*

[prefIntraSsi] Communication preference within a single system image (SSI). *Currently options not documented. Use default.*

[prefInterSsi] Communication preference between different single system images (SSIs). *Currently options not documented. Use default.*

[pthreadMinStackSize] Minimum stack size in byte of any child PET executing as Pthread. By default single threaded child PETs do *not* execute as Pthread, and their stack size is unaffected by this argument. However, for multi-threaded child PETs, or if `forceChildPthreads` is `.true.`, child PETs execute as Pthreads with their own private stack.

For cases where OpenMP threads are used by the user code, each thread allocates its own private stack. For all threads *other* than the master, the stack size is set via the typical `OMP_STACKSIZE` environment variable mechanism. The PET itself, however, becomes the *master* of the OpenMP thread team, and is not affected by `OMP_STACKSIZE`. It is the master's stack that can be sized via the `pthreadMinStackSize` argument, and a large enough size is often critical.

When `pthreadMinStackSize` is absent, the default is to use the system default set by the `limit` or `ulimit` command. However, the stack of a Pthread cannot be unlimited, and a shell *stacksize* setting of *unlimited*, or any setting below the ESMF implemented minimum, will result in setting the stack size to 20MiB (the ESMF minimum). Depending on how much private data is used by the user code under the master thread, the default might be too small, and `pthreadMinStackSize` must be used to allocate sufficient stack space.

[forceChildPthreads] For `.true.`, force each child PET to execute in its own Pthread. By default, `.false.`, single PETs spawned from a parent PET execute in the same thread (or MPI process) as the parent PET. Multiple child PETs spawned by the same parent PET always execute as their own Pthreads.

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

17.4.28 ESMF_GridCompValidate - Check validity of a GridComp

INTERFACE:

```
subroutine ESMF_GridCompValidate(gridcomp, rc)
```

ARGUMENTS:

```
    type(ESMF_GridComp), intent(in)            :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    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:

Currently all this method does is to check that the `gridcomp` was created.

The arguments are:

gridcomp `ESMF_GridComp` to validate.

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

17.4.29 ESMF_GridCompWait - Wait for a GridComp to return

INTERFACE:

```
  subroutine ESMF_GridCompWait(gridcomp, syncflag, &
    timeout, timeoutFlag, userRc, rc)
```

ARGUMENTS:

```
    type(ESMF_GridComp),  intent(inout)         :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_Sync_Flag), intent(in),  optional :: syncflag
    integer,              intent(in),  optional :: timeout
    logical,              intent(out), optional :: timeoutFlag
    integer,              intent(out), optional :: userRc
    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:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

When executing asynchronously, wait for an ESMF_GridComp to return.

The arguments are:

gridcomp ESMF_GridComp to wait for.

[syncflag] Blocking behavior of this method call. See section **??** for a list of valid blocking options. Default option is ESMF_SYNC_VASBLOCKING which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.

[timeout] The maximum period in seconds the actual component is allowed to execute a previously invoked component method before it must communicate back to the dual component. If the actual component does not communicate back in the specified time, a timeout condition is raised on the dual side (this side). The default is 3600, i.e. 1 hour. The timeout argument is only supported for connected dual components.

[timeoutFlag] Returns .true. if the timeout was reached, .false. otherwise. If timeoutFlag was *not* provided, a timeout condition will lead to a return code of rc \= ESMF_SUCCESS. Otherwise the return value of timeoutFlag is the sole indicator of a timeout condition.

[userRc] Return code set by userRoutine before returning.

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

17.4.30 ESMF_GridCompWriteRestart - Call the GridComp's write restart routine

INTERFACE:

```
recursive subroutine ESMF_GridCompWriteRestart(gridcomp, &
  importState, exportState, clock, syncflag, phase, timeout, timeoutFlag, &
  userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_GridComp),  intent(inout)           :: gridcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_State),     intent(inout), optional :: importState
  type(ESMF_State),     intent(inout), optional :: exportState
  type(ESMF_Clock),     intent(inout), optional :: clock
  type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
  integer,              intent(in),    optional :: phase
  integer,              intent(in),    optional :: timeout
  logical,              intent(out),   optional :: timeoutFlag
  integer,              intent(out),   optional :: userRc
  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:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Call the associated user write restart routine for an `ESMF_GridComp`.

The arguments are:

gridcomp `ESMF_GridComp` to call run routine for.

[importState] `ESMF_State` containing import data. If not present, a dummy argument will be passed to the user-supplied routine. The importState argument in the user code cannot be optional.

[exportState] `ESMF_State` containing export data. If not present, a dummy argument will be passed to the user-supplied routine. The exportState argument in the user code cannot be optional.

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

[syncflag] Blocking behavior of this method call. See section **??** for a list of valid blocking options. Default option is `ESMF_SYNC_VASBLOCKING` which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.

[phase] Component providers must document whether each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple sub-routines to accomplish the work, accommodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument is optional. The default is 1.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The `timeout` argument is only supported for connected dual components.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

[userRc] Return code set by `userRoutine` before returning.

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

18 CplComp Class

18.1 Description

In a large, multi-component application such as a weather forecasting or climate prediction system running within ESMF, physical domains and major system functions are represented as Gridded Components (see Section 17.1).

A Coupler Component, or ESMF_CplComp, arranges and executes the data transformations between the Gridded Components. Ideally, Coupler Components should contain all the information about inter-component communication for an application. This enables the Gridded Components in the application to be used in multiple contexts; that is, used in different coupled configurations without changes to their source code. For example, the same atmosphere might in one case be coupled to an ocean in a hurricane prediction model, and to a data assimilation system for numerical weather prediction in another. A single Coupler Component can couple two or more Gridded Components.

Like Gridded Components, Coupler Components have two parts, one that is provided by the user and another that is part of the framework. The user-written portion of the software is the coupling code necessary for a particular exchange between Gridded Components. This portion of the Coupler Component code must be divided into separately callable initialize, run, and finalize methods. The interfaces for these methods are prescribed by ESMF.

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

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

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

18.2 Use and Examples

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

18.2.1 Implement a user-code SetServices routine

Every ESMF_CplComp 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_CplComp 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_CplCompSetEntryPoint() 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 Coupler Component
module ESMF_CouplerEx
```

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

```
! ESMF Framework module
use ESMF
implicit none
public CPL_SetServices

contains

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

  ! Set the entry points for standard ESMF Component methods
  call ESMF_CplCompSetEntryPoint(comp, ESMF_METHOD_INITIALIZE, &
                       userRoutine=CPL_Init, rc=rc)
  call ESMF_CplCompSetEntryPoint(comp, ESMF_METHOD_RUN, &
                       userRoutine=CPL_Run, rc=rc)
  call ESMF_CplCompSetEntryPoint(comp, ESMF_METHOD_FINALIZE, &
                       userRoutine=CPL_Final, rc=rc)

  rc = ESMF_SUCCESS
end subroutine
```

18.2.2 Implement a user-code `Initialize` routine

When a higher level component is ready to begin using an `ESMF_CplComp`, 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 CPL_Init(comp, importState, exportState, clock, rc)
  type(ESMF_CplComp)    :: 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 *, "Coupler Init starting"

  ! Add whatever code here needed
  ! Precompute any needed values, fill in any inital values
  !  needed in Import States

  rc = ESMF_SUCCESS

  print *, "Coupler Init returning"
```

```
end subroutine CPL_Init
```

18.2.3 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_CplComp` 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 CPL_Run(comp, importState, exportState, clock, rc)
  type(ESMF_CplComp)    :: 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 *, "Coupler Run starting"

  ! Add whatever code needed here to transform Export state data
  !  into Import states for the next timestep.

  rc = ESMF_SUCCESS

  print *, "Coupler Run returning"

end subroutine CPL_Run
```

18.2.4 Implement a user-code `Finalize` routine

At the end of application execution, each `ESMF_CplComp` 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 CPL_Final(comp, importState, exportState, clock, rc)
  type(ESMF_CplComp)    :: 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 *, "Coupler Final starting"

  ! Add whatever code needed here to compute final values and
  !  finish the computation.

  rc = ESMF_SUCCESS

  print *, "Coupler Final returning"

end subroutine CPL_Final
```

18.2.5 Implement a user-code `SetVM` routine

Every `ESMF_CplComp` can optionally provide and document a public set vm routine. It can have any name, but must follow the declaration below: a subroutine which takes an `ESMF_CplComp` 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 vm routine is the only place where the child component can use the `ESMF_CplCompSetVMMaxPEs()`, or `ESMF_CplCompSetVMMaxThreads()`, or `ESMF_CplCompSetVMMinThreads()` call to modify aspects of its own VM.

A component's VM is started up right before its set services routine is entered. `ESMF_CplCompSetVM()` is executing in the parent VM, and must be called *before* `ESMF_CplCompSetServices()`.

```
subroutine GComp_SetVM(comp, rc)
  type(ESMF_CplComp)    :: comp   ! must not be optional
  integer, intent(out)  :: rc     ! must not be optional

  type(ESMF_VM) :: vm
  logical :: pthreadsEnabled

  ! Test for Pthread support, all SetVM calls require it
  call ESMF_VMGetGlobal(vm, rc=rc)
  call ESMF_VMGet(vm, pthreadsEnabledFlag=pthreadsEnabled, rc=rc)

  if (pthreadsEnabled) then
    ! run PETs single-threaded
    call ESMF_CplCompSetVMMinThreads(comp, rc=rc)
  endif

  rc = ESMF_SUCCESS

end subroutine

end module ESMF_CouplerEx
```

18.3 Restrictions and Future Work

1. **No optional arguments.** User-written routines called by SetServices, and registered for Initialize, Run and Finalize, *must not* declare any of the arguments as optional.
2. **No Transforms.** Components must exchange data through `ESMF_State` objects. The input data are available at the time the component code is called, and data to be returned to another component are available when that code returns.
3. **No automatic unit conversions.** The ESMF framework does not currently contain tools for performing unit conversions, operations that are fairly standard within Coupler Components.
4. **No accumulator.** The ESMF does not have an accumulator tool, to perform time averaging of fields for coupling. This is likely to be developed in the near term.

18.4 Class API

18.4.1 ESMF_CplCompAssignment(=) - CplComp assignment

INTERFACE:

```
interface assignment(=)
cplcomp1 = cplcomp2
```

ARGUMENTS:

```
type(ESMF_CplComp) :: cplcomp1
type(ESMF_CplComp) :: cplcomp2
```

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:

Assign cplcomp1 as an alias to the same ESMF CplComp object in memory as cplcomp2. If cplcomp2 is invalid, then cplcomp1 will be equally invalid after the assignment.

The arguments are:

cplcomp1 The `ESMF_CplComp` object on the left hand side of the assignment.

cplcomp2 The `ESMF_CplComp` object on the right hand side of the assignment.

18.4.2 ESMF_CplCompOperator(==) - CplComp equality operator

INTERFACE:

```
interface operator(==)
  if (cplcomp1 == cplcomp2) then ... endif
            OR
  result = (cplcomp1 == cplcomp2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_CplComp), intent(in) :: cplcomp1
type(ESMF_CplComp), intent(in) :: cplcomp2
```

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:

Test whether cplcomp1 and cplcomp2 are valid aliases to the same ESMF CplComp object in memory. For a more general comparison of two ESMF CplComps, going beyond the simple alias test, the ESMF_CplCompMatch() function (not yet implemented) must be used.

The arguments are:

cplcomp1 The `ESMF_CplComp` object on the left hand side of the equality operation.

cplcomp2 The `ESMF_CplComp` object on the right hand side of the equality operation.

18.4.3 ESMF_CplCompOperator(/=) - CplComp not equal operator

INTERFACE:

```
interface operator(/=)
  if (cplcomp1 /= cplcomp2) then ... endif
            OR
  result = (cplcomp1 /= cplcomp2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_CplComp), intent(in) :: cplcomp1
type(ESMF_CplComp), intent(in) :: cplcomp2
```

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:

Test whether cplcomp1 and cplcomp2 are *not* valid aliases to the same ESMF CplComp object in memory. For a more general comparison of two ESMF CplComps, going beyond the simple alias test, the ESMF_CplCompMatch() function (not yet implemented) must be used.

The arguments are:

cplcomp1 The `ESMF_CplComp` object on the left hand side of the non-equality operation.

cplcomp2 The `ESMF_CplComp` object on the right hand side of the non-equality operation.

18.4.4 ESMF_CplCompCreate - Create a CplComp

INTERFACE:

```
recursive function ESMF_CplCompCreate(config, configFile, &
  clock, petList, contextflag, name, rc)
```

RETURN VALUE:

```
type(ESMF_CplComp) :: ESMF_CplCompCreate
```

ARGUMENTS:

```
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_Config),       intent(in),  optional :: config
    character(len=*),        intent(in),  optional :: configFile
    type(ESMF_Clock),        intent(in),  optional :: clock
    integer,                 intent(in),  optional :: petList(:)
    type(ESMF_Context_Flag), intent(in),  optional :: contextflag
    character(len=*),        intent(in),  optional :: name
    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 interface creates an ESMF_CplComp object. By default, a separate VM context will be created for each component. This implies creating a new MPI communicator and allocating additional memory to manage the VM resources. When running on a large number of processors, creating a separate VM for each component could be both time and memory inefficient. If the application is sequential, i.e., each component is running on all the PETs of the global VM, it will be more efficient to use the global VM instead of creating a new one. This can be done by setting contextflag to ESMF_CONTEXT_PARENT_VM.

The return value is the new ESMF_CplComp.

The arguments are:

[config] An already-created ESMF_Config object to be attached to the newly created component. If both config and configFile arguments are specified, config takes priority.

[configFile] The filename of an ESMF_Config format file. If specified, a new ESMF_Config object is created and attached to the newly created component. The configFile file is opened and associated with the new config object. If both config and configFile arguments are specified, config takes priority.

[clock] Component-specific ESMF_Clock. This clock is available to be queried and updated by the new ESMF_CplComp as it chooses. This should not be the parent component clock, which should be maintained and passed down to the initialize/run/finalize routines separately.

[petList] List of parent PETs given to the created child component by the parent component. If petList is not specified all of the parent PETs will be given to the child component. The order of PETs in petList determines how the child local PETs refer back to the parent PETs.

[contextflag] Specify the component's VM context. The default context is ESMF_CONTEXT_OWN_VM. See section **??** for a complete list of valid flags.

[name] Name of the newly-created ESMF_CplComp. This name can be altered from within the ESMF_CplComp code once the initialization routine is called.

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

18.4.5 ESMF_CplCompDestroy - Release resources associated with a CplComp

INTERFACE:

```
recursive subroutine ESMF_CplCompDestroy(cplcomp, &
  timeout, timeoutFlag, rc)
```

ARGUMENTS:

```
    type(ESMF_CplComp), intent(inout)          :: cplcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,            intent(in),   optional :: timeout
    logical,            intent(out),  optional :: timeoutFlag
    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:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Destroys an `ESMF_CplComp`, releasing the resources associated with the object.

The arguments are:

cplcomp Release all resources associated with this `ESMF_CplComp` and mark the object as invalid. It is an error to pass this object into any other routines after being destroyed.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The `timeout` argument is only supported for connected dual components.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

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

18.4.6 ESMF_CplCompFinalize - Call the CplComp's finalize routine

INTERFACE:

```
recursive subroutine ESMF_CplCompFinalize(cplcomp, &
  importState, exportState, clock, syncflag, phase, timeout, timeoutFlag, &
  userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp),   intent(inout)           :: cplcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_State),     intent(inout), optional :: importState
  type(ESMF_State),     intent(inout), optional :: exportState
  type(ESMF_Clock),     intent(inout), optional :: clock
  type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
  integer,              intent(in),    optional :: phase
  integer,              intent(in),    optional :: timeout
  logical,              intent(out),   optional :: timeoutFlag
  integer,              intent(out),   optional :: userRc
  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:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Call the associated user-supplied finalization routine for an `ESMF_CplComp`.

The arguments are:

cplcomp The `ESMF_CplComp` to call finalize routine for.

[importState] `ESMF_State` containing import data for coupling. If not present, a dummy argument will be passed to the user-supplied routine. The importState argument in the user code cannot be optional.

[exportState] `ESMF_State` containing export data for coupling. If not present, a dummy argument will be passed to the user-supplied routine. The exportState argument in the user code cannot be optional.

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

[syncflag] Blocking behavior of this method call. See section **??** for a list of valid blocking options. Default option is `ESMF_SYNC_VASBLOCKING` which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.

[phase] Component providers must document whether each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accommodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument is optional. The default is 1.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The `timeout` argument is only supported for connected dual components.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

[userRc] Return code set by `userRoutine` before returning.

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

18.4.7 ESMF_CplCompGet - Get CplComp information

INTERFACE:

```
  subroutine ESMF_CplCompGet(cplcomp, configIsPresent, config, &
    configFileIsPresent, configFile, clockIsPresent, clock, localPet, &
    petCount, contextflag, currentMethod, currentPhase, vmIsPresent, &
    vm, name, rc)
```

ARGUMENTS:

```
    type(ESMF_CplComp),      intent(in)            :: cplcomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    logical,                 intent(out), optional :: configIsPresent
    type(ESMF_Config),       intent(out), optional :: config
    logical,                 intent(out), optional :: configFileIsPresent
    character(len=*),        intent(out), optional :: configFile
    logical,                 intent(out), optional :: clockIsPresent
    type(ESMF_Clock),        intent(out), optional :: clock
    integer,                 intent(out), optional :: localPet
    integer,                 intent(out), optional :: petCount
    type(ESMF_Context_Flag), intent(out), optional :: contextflag
    type(ESMF_Method_Flag),  intent(out), optional :: currentMethod
    integer,                 intent(out), optional :: currentPhase
    logical,                 intent(out), optional :: vmIsPresent
    type(ESMF_VM),           intent(out), optional :: vm
    character(len=*),        intent(out), optional :: name
    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:

Get information about an `ESMF_CplComp` object.

The arguments are:

cplcomp The `ESMF_CplComp` object being queried.

[configIsPresent] `.true.` if `config` was set in CplComp object, `.false.` otherwise.

[config] Return the associated Config. It is an error to query for the Config if none is associated with the CplComp. If unsure, get `configIsPresent` first to determine the status.

[configFileIsPresent] `.true.` if `configFile` was set in CplComp object, `.false.` otherwise.

[configFile] Return the associated configuration filename. It is an error to query for the configuration filename if none is associated with the CplComp. If unsure, get `configFileIsPresent` first to determine the status.

[clockIsPresent] `.true.` if `clock` was set in CplComp object, `.false.` otherwise.

[clock] Return the associated Clock. It is an error to query for the Clock if none is associated with the CplComp. If unsure, get `clockIsPresent` first to determine the status.

[localPet] Return the local PET id within the `ESMF_CplComp` object.

[petCount] Return the number of PETs in the the `ESMF_CplComp` object.

[contextflag] Return the `ESMF_Context_Flag` for this `ESMF_CplComp`. See section **??** for a complete list of valid flags.

[currentMethod] Return the current `ESMF_Method_Flag` of the `ESMF_CplComp` execution. See section **??** for a complete list of valid options.

[currentPhase] Return the current `phase` of the `ESMF_CplComp` execution.

[vmIsPresent] `.true.` if `vm` was set in CplComp object, `.false.` otherwise.

[vm] Return the associated VM. It is an error to query for the VM if none is associated with the CplComp. If unsure, get `vmIsPresent` first to determine the status.

[name] Return the name of the `ESMF_CplComp`.

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

18.4.8 ESMF_CplCompGetInternalState - Get private data block pointer

INTERFACE:

```
subroutine ESMF_CplCompGetInternalState(cplcomp, wrappedDataPointer, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)              :: cplcomp
type(wrapper)                   :: wrappedDataPointer
integer,            intent(out) :: 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:

Available to be called by an `ESMF_CplComp` at any time after `ESMF_CplCompSetInternalState` has been called. Since init, run, and finalize must be separate subroutines, data that they need to share in common can either be module global data, or can be allocated in a private data block and the address of that block can be registered with the framework and retrieved by this call. When running multiple instantiations of an `ESMF_CplComp`, for example during ensemble runs, it may be simpler to maintain private data specific to each run with private data blocks. A corresponding `ESMF_CplCompSetInternalState` call sets the data pointer to this block, and this call retrieves the data pointer. Note that the `wrappedDataPointer` argument needs to be a derived type which contains only a pointer of the type of the data block defined by the user. When making this call the pointer needs to be unassociated.

When the call returns, the pointer will now reference the original data block which was set during the previous call to ESMF_CplCompSetInternalState.

Only the *last* data block set via ESMF_CplCompSetInternalState will be accessible.

CAUTION: If you are working with a compiler that does not support Fortran 2018 assumed-type dummy arguments, then this method does not have an explicit Fortran interface. In this case do not specify argument keywords when calling this method!

The arguments are:

cplcomp An ESMF_CplComp object.

wrappedDataPointer A derived type (wrapper), containing only an unassociated pointer to the private data block. The framework will fill in the pointer. When this call returns, the pointer is set to the same address set during the last ESMF_CplCompSetInternalState call. This level of indirection is needed to reliably set and retrieve the data block no matter which architecture or compiler is used.

rc Return code; equals ESMF_SUCCESS if there are no errors. Note: unlike most other ESMF routines, this argument is not optional because of implementation considerations.

18.4.9 ESMF_CplCompInitialize - Call the CplComp's initialize routine

INTERFACE:

```
  recursive subroutine ESMF_CplCompInitialize(cplcomp, &
    importState, exportState, clock, syncflag, phase, timeout, timeoutFlag, &
    userRc, rc)
```

ARGUMENTS:

```
    type(ESMF_CplComp),   intent(inout)           :: cplcomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_State),     intent(inout), optional :: importState
    type(ESMF_State),     intent(inout), optional :: exportState
    type(ESMF_Clock),     intent(inout), optional :: clock
    type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
    integer,              intent(in),    optional :: phase
    integer,              intent(in),    optional :: timeout
    logical,              intent(out),   optional :: timeoutFlag
    integer,              intent(out),   optional :: userRc
    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:

5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Call the associated user initialization routine for an `ESMF_CplComp`.

The arguments are:

cplcomp `ESMF_CplComp` to call initialize routine for.

[importState] `ESMF_State` containing import data for coupling. If not present, a dummy argument will be passed to the user-supplied routine. The importState argument in the user code cannot be optional.

[exportState] `ESMF_State` containing export data for coupling. If not present, a dummy argument will be passed to the user-supplied routine. The exportState argument in the user code cannot be optional.

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

[syncflag] Blocking behavior of this method call. See section **??** for a list of valid blocking options. Default option is `ESMF_SYNC_VASBLOCKING` which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.

[phase] Component providers must document whether each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accommodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument is optional. The default is 1.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The `timeout` argument is only supported for connected dual components.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

[userRc] Return code set by `userRoutine` before returning.

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

18.4.10 ESMF_CplCompIsCreated - Check whether a CplComp object has been created

INTERFACE:

```
function ESMF_CplCompIsCreated(cplcomp, rc)
```

RETURN VALUE:

```
    logical :: ESMF_CplCompIsCreated
```

ARGUMENTS:

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

DESCRIPTION:

Return `.true.` if the `cplcomp` has been created. Otherwise return `.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:

cplcomp `ESMF_CplComp` queried.

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

18.4.11 ESMF_CplCompIsPetLocal - Inquire if this CplComp is to execute on the calling PET

INTERFACE:

```
  recursive function ESMF_CplCompIsPetLocal(cplcomp, rc)
```

RETURN VALUE:

```
    logical :: ESMF_CplCompIsPetLocal
```

ARGUMENTS:

```
    type(ESMF_CplComp), intent(in)            :: cplcomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    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:

Inquire if this `ESMF_CplComp` object is to execute on the calling PET.

The return value is `.true.` if the component is to execute on the calling PET, `.false.` otherwise.

The arguments are:

cplcomp `ESMF_CplComp` queried.

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

18.4.12 ESMF_CplCompPrint - Print CplComp information

INTERFACE:

```
subroutine ESMF_CplCompPrint(cplcomp, rc)
```

ARGUMENTS:

```
    type(ESMF_CplComp), intent(in)            :: cplcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    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:

Prints information about an `ESMF_CplComp` to `stdout`.

The arguments are:

cplcomp `ESMF_CplComp` to print.

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

18.4.13 ESMF_CplCompReadRestart – Call the CplComp's read restart routine

INTERFACE:

```
recursive subroutine ESMF_CplCompReadRestart(cplcomp, &
  importState, exportState, clock, syncflag, phase, timeout, timeoutFlag, &
  userRc, rc)
```

ARGUMENTS:

```
    type(ESMF_CplComp),   intent(inout)           :: cplcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_State),     intent(inout), optional :: importState
    type(ESMF_State),     intent(inout), optional :: exportState
    type(ESMF_Clock),     intent(inout), optional :: clock
    type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
    integer,              intent(in),    optional :: phase
    integer,              intent(in),    optional :: timeout
    logical,              intent(out),   optional :: timeoutFlag
    integer,              intent(out),   optional :: userRc
    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:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Call the associated user read restart routine for an `ESMF_CplComp`.

The arguments are:

cplcomp `ESMF_CplComp` to call run routine for.

[importState] `ESMF_State` containing import data. If not present, a dummy argument will be passed to the user-supplied routine. The importState argument in the user code cannot be optional.

[exportState] `ESMF_State` containing export data. If not present, a dummy argument will be passed to the user-supplied routine. The exportState argument in the user code cannot be optional.

[clock] External `ESMF_Clock` for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

[syncflag] Blocking behavior of this method call. See section **??** for a list of valid blocking options. Default option is `ESMF_SYNC_VASBLOCKING` which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.

[phase] Component providers must document whether each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accommodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument is optional. The default is 1.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The `timeout` argument is only supported for connected dual components.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

[userRc] Return code set by `userRoutine` before returning.

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

18.4.14 ESMF_CplCompRun - Call the CplComp's run routine

INTERFACE:

```
recursive subroutine ESMF_CplCompRun(cplcomp, &
  importState, exportState, clock, syncflag, phase, timeout, timeoutFlag, &
  userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp),   intent(inout)           :: cplcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_State),     intent(inout), optional :: importState
  type(ESMF_State),     intent(inout), optional :: exportState
  type(ESMF_Clock),     intent(inout), optional :: clock
  type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
  integer,              intent(in),    optional :: phase
  integer,              intent(in),    optional :: timeout
  logical,              intent(out),   optional :: timeoutFlag
  integer,              intent(out),   optional :: userRc
  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:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Call the associated user run routine for an `ESMF_CplComp`.

The arguments are:

cplcomp ESMF_CplComp to call run routine for.

[importState] ESMF_State containing import data for coupling. If not present, a dummy argument will be passed to the user-supplied routine. The importState argument in the user code cannot be optional.

[exportState] ESMF_State containing export data for coupling. If not present, a dummy argument will be passed to the user-supplied routine. The exportState argument in the user code cannot be optional.

[clock] External ESMF_Clock for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

[syncflag] Blocking behavior of this method call. See section **??** for a list of valid blocking options. Default option is ESMF_SYNC_VASBLOCKING which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.

[phase] Component providers must document whether each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accommodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument is optional. The default is 1.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The timeout argument is only supported for connected dual components.

[timeoutFlag] Returns .true. if the timeout was reached, .false. otherwise. If timeoutFlag was *not* provided, a timeout condition will lead to a return code of rc \= ESMF_SUCCESS. Otherwise the return value of timeoutFlag is the sole indicator of a timeout condition.

[userRc] Return code set by userRoutine before returning.

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

18.4.15 ESMF_CplCompServiceLoop - Call the CplComp's service loop routine

INTERFACE:

```
recursive subroutine ESMF_CplCompServiceLoop(cplcomp, &
  importState, exportState, clock, syncflag, port, timeout, timeoutFlag, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp),   intent(inout)           :: cplcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_State),     intent(inout), optional :: importState
  type(ESMF_State),     intent(inout), optional :: exportState
  type(ESMF_Clock),     intent(inout), optional :: clock
  type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
  integer,              intent(in),    optional :: port
  integer,              intent(in),    optional :: timeout
  logical,              intent(out),   optional :: timeoutFlag
  integer,              intent(out),   optional :: rc
```

DESCRIPTION:

Call the ServiceLoop routine for an ESMF_CplComp. This tries to establish a "component tunnel" between the *actual* Component (calling this routine) and a dual Component connecting to it through a matching SetServices call.

The arguments are:

cplcomp ESMF_CplComp to call service loop routine for.

[importState] ESMF_State containing import data for coupling. If not present, a dummy argument will be passed to the user-supplied routine. The importState argument in the user code cannot be optional.

[exportState] ESMF_State containing export data for coupling. If not present, a dummy argument will be passed to the user-supplied routine. The exportState argument in the user code cannot be optional.

[clock] External ESMF_Clock for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

[syncflag] Blocking behavior of this method call. See section **??** for a list of valid blocking options. Default option is ESMF_SYNC_VASBLOCKING which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.

[port] In case a port number is provided, the "component tunnel" is established using sockets. The actual component side, i.e. the side that calls into ESMF_CplCompServiceLoop(), starts to listen on the specified port as the server. The valid port range is [1024, 65535]. In case the port argument is *not* specified, the "component tunnel" is established within the same executable using local communication methods (e.g. MPI).

[timeout] The maximum period in seconds that this call will wait for communications with the dual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. (NOTE: Currently this option is only available for socket based component tunnels.)

[timeoutFlag] Returns .true. if the timeout was reached, .false. otherwise. If timeoutFlag was *not* provided, a timeout condition will lead to a return code of rc \= ESMF_SUCCESS. Otherwise the return value of timeoutFlag is the sole indicator of a timeout condition.

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

18.4.16 ESMF_CplCompSet - Set or reset information about the CplComp

INTERFACE:

```
  subroutine ESMF_CplCompSet(cplcomp, config, configFile, &
    clock, name, rc)
```

ARGUMENTS:

```
    type(ESMF_CplComp), intent(inout)         :: cplcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_Config),  intent(in),  optional :: config
    character(len=*),   intent(in),  optional :: configFile
```

```
type(ESMF_Clock),   intent(in),  optional :: clock
character(len=*),   intent(in),  optional :: name
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:

Sets or resets information about an ESMF_CplComp.

The arguments are:

cplcomp ESMF_CplComp to change.

[name] Set the name of the ESMF_CplComp.

[config] An already-created ESMF_Config object to be attached to the component. If both config and configFile arguments are specified, config takes priority.

[configFile] The filename of an ESMF_Config format file. If specified, a new ESMF_Config object is created and attached to the component. The configFile file is opened and associated with the new config object. If both config and configFile arguments are specified, config takes priority.

[clock] Set the private clock for this ESMF_CplComp.

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

18.4.17 ESMF_CplCompSetEntryPoint - Set user routine as entry point for standard Component method

INTERFACE:

```
recursive subroutine ESMF_CplCompSetEntryPoint(cplcomp, methodflag, &
  userRoutine, phase, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp),     intent(inout)         :: cplcomp
type(ESMF_Method_Flag), intent(in)            :: methodflag
interface
  subroutine userRoutine(cplcomp, importState, exportState, clock, rc)
    use ESMF_CompMod
    use ESMF_StateMod
    use ESMF_ClockMod
    implicit none
    type(ESMF_CplComp)          :: cplcomp      ! 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
      end subroutine
    end interface
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,                intent(in),  optional :: phase
    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:

Registers a user-supplied `userRoutine` as the entry point for one of the predefined Component `methodflag`s. After this call the `userRoutine` becomes accessible via the standard Component method API.

The arguments are:

cplcomp An `ESMF_CplComp` object.

methodflag One of a set of predefined Component methods - e.g. `ESMF_METHOD_INITIALIZE`, `ESMF_METHOD_RUN`, `ESMF_METHOD_FINALIZE`. See section **??** for a complete list of valid method options.

userRoutine The user-supplied subroutine to be associated for this `methodflag`. The Component writer must supply a subroutine with the exact interface shown above for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

[phase] The `phase` number for multi-phase methods. For single phase methods the `phase` argument can be omitted. The default setting is 1.

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

18.4.18 ESMF_CplCompSetInternalState - Set private data block pointer

INTERFACE:

```
subroutine ESMF_CplCompSetInternalState(cplcomp, wrappedDataPointer, rc)
```

ARGUMENTS:

```
    type(ESMF_CplComp)              :: cplcomp
    type(wrapper)                   :: wrappedDataPointer
    integer,            intent(out) :: 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:

Available to be called by an ESMF_CplComp at any time, but expected to be most useful when called during the registration process, or initialization. Since init, run, and finalize must be separate subroutines data that they need to share in common can either be module global data, or can be allocated in a private data block and the address of that block can be registered with the framework and retrieved by subsequent calls. When running multiple instantiations of an ESMF_CplComp, for example during ensemble runs, it may be simpler to maintain private data specific to each run with private data blocks. A corresponding ESMF_CplCompGetInternalState call retrieves the data pointer.

Only the *last* data block set via ESMF_CplCompSetInternalState will be accessible.

CAUTION: If you are working with a compiler that does not support Fortran 2018 assumed-type dummy arguments, then this method does not have an explicit Fortran interface. In this case do not specify argument keywords when calling this method!

The arguments are:

cplcomp An ESMF_CplComp object.

wrappedDataPointer A pointer to the private data block, wrapped in a derived type which contains only a pointer to the block. This level of indirection is needed to reliably set and retrieve the data block no matter which architecture or compiler is used.

rc Return code; equals ESMF_SUCCESS if there are no errors. Note: unlike most other ESMF routines, this argument is not optional because of implementation considerations.

18.4.19 ESMF_CplCompSetServices - Call user routine to register CplComp methods

INTERFACE:

```
recursive subroutine ESMF_CplCompSetServices(cplcomp, userRoutine, &
   userRc, rc)
```

ARGUMENTS:

```
    type(ESMF_CplComp), intent(inout)         :: cplcomp
    interface
      subroutine userRoutine(cplcomp, rc)
        use ESMF_CompMod
        implicit none
        type(ESMF_CplComp)         :: cplcomp  ! must not be optional
        integer, intent(out)       :: rc       ! must not be optional
      end subroutine
    end interface
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,            intent(out), optional :: userRc
    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:

Call into user provided `userRoutine` which is responsible for setting Component's Initialize(), Run(), and Finalize() services.

The arguments are:

cplcomp Coupler Component.

userRoutine The Component writer must supply a subroutine with the exact interface shown above for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

The `userRoutine`, when called by the framework, must make successive calls to `ESMF_CplCompSetEntryPoint()` to preset callback routines for standard Component Initialize(), Run(), and Finalize() methods.

[userRc] Return code set by `userRoutine` before returning.

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

18.4.20 ESMF_CplCompSetServices - Call user routine through name lookup, to register CplComp methods

INTERFACE:

```
! Private name; call using ESMF_CplCompSetServices()
recursive subroutine ESMF_CplCompSetServicesShObj(cplcomp, userRoutine, &
  sharedObj, userRoutineFound, userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp),  intent(inout)          :: cplcomp
  character(len=*),    intent(in)             :: userRoutine
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  character(len=*),    intent(in),  optional :: sharedObj
  logical,             intent(out), optional :: userRoutineFound
  integer,             intent(out), optional :: userRc
  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:

  6.3.0r Added argument `userRoutineFound`. The new argument provides a way to test availability without causing error conditions.

DESCRIPTION:

Call into a user provided routine which is responsible for setting Component's Initialize(), Run(), and Finalize() services. The named `userRoutine` must exist in the executable, or in the shared object specified by `sharedObj`. In the latter case all of the platform specific details about dynamic linking and loading apply.

The arguments are:

cplcomp Coupler Component.

userRoutine Name of routine to be called, specified as a character string. The Component writer must supply a subroutine with the exact interface shown for `userRoutine` below. Arguments must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

INTERFACE:

```
interface
  subroutine userRoutine(cplcomp, rc)
    type(ESMF_CplComp)   :: cplcomp    ! must not be optional
    integer, intent(out) :: rc         ! must not be optional
  end subroutine
end interface
```

DESCRIPTION:

The `userRoutine`, when called by the framework, must make successive calls to `ESMF_CplCompSetEntryPoint()` to preset callback routines for standard Component Initialize(), Run(), and Finalize() methods.

[sharedObj] Name of shared object that contains `userRoutine`. If the `sharedObj` argument is not provided the executable itself will be searched for `userRoutine`.

[userRoutineFound] Report back whether the specified `userRoutine` was found and executed, or was not available. If this argument is present, not finding the `userRoutine` will not result in returning an error in `rc`. The default is to return an error if the `userRoutine` cannot be found.

[userRc] Return code set by `userRoutine` before returning.

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

18.4.21 ESMF_CplCompSetServices - Set to serve as Dual Component for an Actual Component

INTERFACE:

```
   ! Private name; call using ESMF_CplCompSetServices()
   recursive subroutine ESMF_CplCompSetServicesComp(cplcomp, &
     actualCplcomp, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Set the services of a Coupler Component to serve a "dual" Component for an "actual" Component. The component tunnel is VM based.

The arguments are:

cplcomp Dual Coupler Component.

actualCplcomp Actual Coupler Component.

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

18.4.22 ESMF_CplCompSetServices - Set to serve as Dual Component for an Actual Component through sockets

INTERFACE:

```
   ! Private name; call using ESMF_CplCompSetServices()
   recursive subroutine ESMF_CplCompSetServicesSock(cplcomp, port, &
     server, timeout, timeoutFlag, rc)
```

ARGUMENTS:

```
     type(ESMF_CplComp), intent(inout)         :: cplcomp
     integer,            intent(in)            :: port
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     character(len=*),   intent(in),  optional :: server
     integer,            intent(in),  optional :: timeout
     logical,            intent(out), optional :: timeoutFlag
     integer,            intent(out), optional :: rc
```

DESCRIPTION:

Set the services of a Coupler Component to serve a "dual" Component for an "actual" Component. The component tunnel is socket based.

The arguments are:

cplcomp Dual Coupler Component.

port Port number under which the actual component is being served. The valid port range is [1024, 65535].

[server] Server name where the actual component is being served. The default, i.e. if the `server` argument was not provided, is `localhost`.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

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

18.4.23 ESMF_CplCompSetVM - Call user routine to set CplComp VM properties

INTERFACE:

```
recursive subroutine ESMF_CplCompSetVM(cplcomp, userRoutine, &
  userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp), intent(inout)         :: cplcomp
  interface
    subroutine userRoutine(cplcomp, rc)
      use ESMF_CompMod
      implicit none
      type(ESMF_CplComp)         :: cplcomp  ! must not be optional
      integer, intent(out)       :: rc       ! must not be optional
    end subroutine
  end interface
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  integer,            intent(out), optional :: userRc
  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:

Optionally call into user provided `userRoutine` which is responsible for setting Component's VM properties.

The arguments are:

cplcomp Coupler Component.

userRoutine The Component writer must supply a subroutine with the exact interface shown above for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

The subroutine, when called by the framework, is expected to use any of the `ESMF_CplCompSetVMxxx()` methods to set the properties of the VM associated with the Coupler Component.

[userRc] Return code set by `userRoutine` before returning.

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

18.4.24 ESMF_CplCompSetVM - Call user routine through name lookup, to set CplComp VM properties

INTERFACE:

```
! Private name; call using ESMF_CplCompSetVM()
recursive subroutine ESMF_CplCompSetVMShObj(cplcomp, userRoutine, &
  sharedObj, userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp),  intent(inout)          :: cplcomp
  character(len=*),    intent(in)             :: userRoutine
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  character(len=*),    intent(in),  optional :: sharedObj
  integer,             intent(out), optional :: userRc
  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:

Optionally call into user provided `userRoutine` which is responsible for setting Component's VM properties. The named `userRoutine` must exist in the executable, or in the shared object specified by `sharedObj`. In the latter case all of the platform specific details about dynamic linking and loading apply.

The arguments are:

cplcomp Coupler Component.

userRoutine Routine to be called, specified as a character string. The Component writer must supply a subroutine with the exact interface shown for `userRoutine` below. Arguments must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

INTERFACE:

```
interface
  subroutine userRoutine(cplcomp, rc)
    type(ESMF_CplComp)   :: cplcomp     ! must not be optional
    integer, intent(out) :: rc          ! must not be optional
  end subroutine
end interface
```

DESCRIPTION:

The subroutine, when called by the framework, is expected to use any of the `ESMF_CplCompSetVMxxx()` methods to set the properties of the VM associated with the Coupler Component.

[sharedObj] Name of shared object that contains `userRoutine`. If the `sharedObj` argument is not provided the executable itself will be searched for `userRoutine`.

[userRc] Return code set by `userRoutine` before returning.

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

18.4.25 ESMF_CplCompSetVMMaxPEs - Associate PEs with PETs in CplComp VM

INTERFACE:

```
subroutine ESMF_CplCompSetVMMaxPEs(cplcomp, &
  maxPeCountPerPet, prefIntraProcess, prefIntraSsi, prefInterSsi, &
  pthreadMinStackSize, forceChildPthreads, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp),  intent(inout)         :: cplcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  integer,             intent(in),  optional :: maxPeCountPerPet
  integer,             intent(in),  optional :: prefIntraProcess
  integer,             intent(in),  optional :: prefIntraSsi
  integer,             intent(in),  optional :: prefInterSsi
  integer,             intent(in),  optional :: pthreadMinStackSize
  logical,             intent(in),  optional :: forceChildPthreads
  integer,             intent(out), optional :: rc
```

DESCRIPTION:

Set characteristics of the ESMF_VM for this ESMF_CplComp. Attempts to associate up to maxPeCountPerPet PEs with each PET. Only PEs that are located on the same single system image (SSI) can be associated with the same PET. Within this constraint the call tries to get as close as possible to the number specified by maxPeCountPerPet.

The other constraint to this call is that the number of PEs is preserved. This means that the child Component in the end is associated with as many PEs as the parent Component provided to the child. The number of child PETs however is adjusted according to the above rule.

The typical use of ESMF_CplCompSetVMMaxPEs() is to allocate multiple PEs per PET in a Component for user-level threading, e.g. OpenMP.

The arguments are:

cplcomp ESMF_CplComp to set the ESMF_VM for.

[maxPeCountPerPet] Maximum number of PEs on each PET. Default for each SSI is the local number of PEs.

[prefIntraProcess] Communication preference within a single process. *Currently options not documented. Use default.*

[prefIntraSsi] Communication preference within a single system image (SSI). *Currently options not documented. Use default.*

[prefInterSsi] Communication preference between different single system images (SSIs). *Currently options not documented. Use default.*

[pthreadMinStackSize] Minimum stack size in byte of any child PET executing as Pthread. By default single threaded child PETs do *not* execute as Pthread, and their stack size is unaffected by this argument. However, for multi-threaded child PETs, or if forceChildPthreads is .true., child PETs execute as Pthreads with their own private stack.

For cases where OpenMP threads are used by the user code, each thread allocates its own private stack. For all threads *other* than the master, the stack size is set via the typical OMP_STACKSIZE environment variable mechanism. The PET itself, however, becomes the *master* of the OpenMP thread team, and is not affected by OMP_STACKSIZE. It is the master's stack that can be sized via the pthreadMinStackSize argument, and a large enough size is often critical.

When pthreadMinStackSize is absent, the default is to use the system default set by the limit or ulimit command. However, the stack of a Pthread cannot be unlimited, and a shell *stacksize* setting of *unlimited*, or any setting below the ESMF implemented minimum, will result in setting the stack size to 20MiB (the ESMF minimum). Depending on how much private data is used by the user code under the master thread, the default might be too small, and pthreadMinStackSize must be used to allocate sufficient stack space.

[forceChildPthreads] For .true., force each child PET to execute in its own Pthread. By default, .false., single PETs spawned from a parent PET execute in the same thread (or MPI process) as the parent PET. Multiple child PETs spawned by the same parent PET always execute as their own Pthreads.

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

18.4.26 ESMF_CplCompSetVMMaxThreads - Set multi-threaded PETs in CplComp VM

INTERFACE:

```
  subroutine ESMF_CplCompSetVMMaxThreads(cplcomp, &
    maxPetCountPerVas, prefIntraProcess, prefIntraSsi, prefInterSsi, &
    pthreadMinStackSize, forceChildPthreads, rc)
```

ARGUMENTS:

```
    type(ESMF_CplComp),  intent(inout)         :: cplcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,             intent(in),  optional :: maxPetCountPerVas
    integer,             intent(in),  optional :: prefIntraProcess
    integer,             intent(in),  optional :: prefIntraSsi
    integer,             intent(in),  optional :: prefInterSsi
    integer,             intent(in),  optional :: pthreadMinStackSize
    logical,             intent(in),  optional :: forceChildPthreads
    integer,             intent(out), optional :: rc
```

DESCRIPTION:

Set characteristics of the `ESMF_VM` for this `ESMF_CplComp`. Attempts to provide `maxPetCountPerVas` threaded PETs in each virtual address space (VAS). Only as many threaded PETs as there are PEs located on the single system image (SSI) can be associated with the VAS. Within this constraint the call tries to get as close as possible to the number specified by `maxPetCountPerVas`.

The other constraint to this call is that the number of PETs is preserved. This means that the child Component in the end is associated with as many PETs as the parent Component provided to the child. The threading level of the child PETs however is adjusted according to the above rule.

The typical use of `ESMF_CplCompSetVMMaxThreads()` is to run a Component multi-threaded with groups of PETs executing within a common virtual address space.

The arguments are:

cplcomp `ESMF_CplComp` to set the `ESMF_VM` for.

[maxPetCountPerVas] Maximum number of threaded PETs in each virtual address space (VAS). Default for each SSI is the local number of PEs.

[prefIntraProcess] Communication preference within a single process. *Currently options not documented. Use default.*

[prefIntraSsi] Communication preference within a single system image (SSI). *Currently options not documented. Use default.*

[prefInterSsi] Communication preference between different single system images (SSIs). *Currently options not documented. Use default.*

[pthreadMinStackSize] Minimum stack size in byte of any child PET executing as Pthread. By default single threaded child PETs do *not* execute as Pthread, and their stack size is unaffected by this argument. However, for multi-threaded child PETs, or if `forceChildPthreads` is `.true.`, child PETs execute as Pthreads with their own private stack.

For cases where OpenMP threads are used by the user code, each thread allocates its own private stack. For all threads *other* than the master, the stack size is set via the typical `OMP_STACKSIZE` environment variable mechanism. The PET itself, however, becomes the *master* of the OpenMP thread team, and is not affected by `OMP_STACKSIZE`. It is the master's stack that can be sized via the `pthreadMinStackSize` argument, and a large enough size is often critical.

When pthreadMinStackSize is absent, the default is to use the system default set by the limit or ulimit command. However, the stack of a Pthread cannot be unlimited, and a shell *stacksize* setting of *unlimited*, or any setting below the ESMF implemented minimum, will result in setting the stack size to 20MiB (the ESMF minimum). Depending on how much private data is used by the user code under the master thread, the default might be too small, and pthreadMinStackSize must be used to allocate sufficient stack space.

[forceChildPthreads] For .true., force each child PET to execute in its own Pthread. By default, .false., single PETs spawned from a parent PET execute in the same thread (or MPI process) as the parent PET. Multiple child PETs spawned by the same parent PET always execute as their own Pthreads.

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

18.4.27 ESMF_CplCompSetVMMinThreads - Set a reduced threading level in CplComp VM

INTERFACE:

```
   subroutine ESMF_CplCompSetVMMinThreads(cplcomp, &
     maxPeCountPerPet, prefIntraProcess, prefIntraSsi, prefInterSsi, &
     pthreadMinStackSize, forceChildPthreads, rc)
```

ARGUMENTS:

```
     type(ESMF_CplComp),  intent(inout)         :: cplcomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,             intent(in),  optional :: maxPeCountPerPet
     integer,             intent(in),  optional :: prefIntraProcess
     integer,             intent(in),  optional :: prefIntraSsi
     integer,             intent(in),  optional :: prefInterSsi
     integer,             intent(in),  optional :: pthreadMinStackSize
     logical,             intent(in),  optional :: forceChildPthreads
     integer,             intent(out), optional :: rc
```

DESCRIPTION:

Set characteristics of the ESMF_VM for this ESMF_CplComp. Reduces the number of threaded PETs in each VAS. The max argument may be specified to limit the maximum number of PEs that a single PET can be associated with.

Several constraints apply: 1) the number of PEs cannot change, 2) PEs cannot migrate between single system images (SSIs), 3) the number of PETs cannot increase, only decrease, 4) PETs cannot migrate between virtual address spaces (VASs), nor can VASs migrate between SSIs.

The typical use of ESMF_CplCompSetVMMinThreads() is to run a Component across a set of single-threaded PETs.

The arguments are:

cplcomp ESMF_CplComp to set the ESMF_VM for.

[maxPeCountPerPet] Maximum number of PEs on each PET. Default for each SSI is the local number of PEs.

[prefIntraProcess] Communication preference within a single process. *Currently options not documented. Use default.*

[prefIntraSsi] Communication preference within a single system image (SSI). *Currently options not documented. Use default.*

[prefInterSsi] Communication preference between different single system images (SSIs). *Currently options not documented. Use default.*

[pthreadMinStackSize] Minimum stack size in byte of any child PET executing as Pthread. By default single threaded child PETs do *not* execute as Pthread, and their stack size is unaffected by this argument. However, for multi-threaded child PETs, or if `forceChildPthreads` is `.true.`, child PETs execute as Pthreads with their own private stack.

For cases where OpenMP threads are used by the user code, each thread allocates its own private stack. For all threads *other* than the master, the stack size is set via the typical `OMP_STACKSIZE` environment variable mechanism. The PET itself, however, becomes the *master* of the OpenMP thread team, and is not affected by `OMP_STACKSIZE`. It is the master's stack that can be sized via the `pthreadMinStackSize` argument, and a large enough size is often critical.

When `pthreadMinStackSize` is absent, the default is to use the system default set by the `limit` or `ulimit` command. However, the stack of a Pthread cannot be unlimited, and a shell *stacksize* setting of *unlimited*, or any setting below the ESMF implemented minimum, will result in setting the stack size to 20MiB (the ESMF minimum). Depending on how much private data is used by the user code under the master thread, the default might be too small, and `pthreadMinStackSize` must be used to allocate sufficient stack space.

[forceChildPthreads] For `.true.`, force each child PET to execute in its own Pthread. By default, `.false.`, single PETs spawned from a parent PET execute in the same thread (or MPI process) as the parent PET. Multiple child PETs spawned by the same parent PET always execute as their own Pthreads.

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

18.4.28 ESMF_CplCompValidate – Ensure the CplComp is internally consistent

INTERFACE:

```
subroutine ESMF_CplCompValidate(cplcomp, rc)
```

ARGUMENTS:

```
     type(ESMF_CplComp), intent(in)             :: cplcomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     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:

Currently all this method does is to check that the `cplcomp` was created.

The arguments are:

cplcomp ESMF_CplComp to validate.

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

18.4.29 ESMF_CplCompWait - Wait for a CplComp to return

INTERFACE:

```
   subroutine ESMF_CplCompWait(cplcomp, syncflag, &
     timeout, timeoutFlag, userRc, rc)
```

ARGUMENTS:

```
     type(ESMF_CplComp),   intent(inout)         :: cplcomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_Sync_Flag), intent(in),  optional :: syncflag
     integer,              intent(in),  optional :: timeout
     logical,              intent(out), optional :: timeoutFlag
     integer,              intent(out), optional :: userRc
     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:

  5.3.0 Added argument timeout. Added argument timeoutFlag. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

When executing asynchronously, wait for an ESMF_CplComp to return.

The arguments are:

cplcomp ESMF_CplComp to wait for.

[syncflag] Blocking behavior of this method call. See section **??** for a list of valid blocking options. Default option is ESMF_SYNC_VASBLOCKING which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.

[timeout] The maximum period in seconds the actual component is allowed to execute a previously invoked component method before it must communicate back to the dual component. If the actual component does not communicate back in the specified time, a timeout condition is raised on the dual side (this side). The default is 3600, i.e. 1 hour. The timeout argument is only supported for connected dual components.

[timeoutFlag] Returns `.true.` if the timeout was reached, `.false.` otherwise. If `timeoutFlag` was *not* provided, a timeout condition will lead to a return code of `rc \= ESMF_SUCCESS`. Otherwise the return value of `timeoutFlag` is the sole indicator of a timeout condition.

[userRc] Return code set by `userRoutine` before returning.

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

18.4.30 ESMF_CplCompWriteRestart – Call the CplComp's write restart routine

INTERFACE:

```
recursive subroutine ESMF_CplCompWriteRestart(cplcomp, &
  importState, exportState, clock, syncflag, phase, timeout, timeoutFlag, &
  userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp),   intent(inout)           :: cplcomp
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_State),     intent(inout), optional :: importState
  type(ESMF_State),     intent(inout), optional :: exportState
  type(ESMF_Clock),     intent(inout), optional :: clock
  type(ESMF_Sync_Flag), intent(in),    optional :: syncflag
  integer,              intent(in),    optional :: phase
  integer,              intent(in),    optional :: timeout
  logical,              intent(out),   optional :: timeoutFlag
  integer,              intent(out),   optional :: userRc
  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:

  5.3.0 Added argument `timeout`. Added argument `timeoutFlag`. The new arguments provide access to the fault-tolerant component features.

DESCRIPTION:

Call the associated user write restart routine for an `ESMF_CplComp`.

The arguments are:

cplcomp `ESMF_CplComp` to call run routine for.

[importState] ESMF_State containing import data. If not present, a dummy argument will be passed to the user-supplied routine. The importState argument in the user code cannot be optional.

[exportState] ESMF_State containing export data. If not present, a dummy argument will be passed to the user-supplied routine. The exportState argument in the user code cannot be optional.

[clock] External ESMF_Clock for passing in time information. This is generally the parent component's clock, and will be treated as read-only by the child component. The child component can maintain a private clock for its own internal time computations. If not present, a dummy argument will be passed to the user-supplied routine. The clock argument in the user code cannot be optional.

[syncflag] Blocking behavior of this method call. See section **??** for a list of valid blocking options. Default option is ESMF_SYNC_VASBLOCKING which blocks PETs and their spawned off threads across each VAS but does not synchronize PETs that run in different VASs.

[phase] Component providers must document whether each of their routines are *single-phase* or *multi-phase*. Single-phase routines require only one invocation to complete their work. Multi-phase routines provide multiple subroutines to accomplish the work, accommodating components which must complete part of their work, return to the caller and allow other processing to occur, and then continue the original operation. For multiple-phase child components, this is the integer phase number to be invoked. For single-phase child components this argument is optional. The default is 1.

[timeout] The maximum period in seconds that this call will wait in communications with the actual component, before returning with a timeout condition. The default is 3600, i.e. 1 hour. The timeout argument is only supported for connected dual components.

[timeoutFlag] Returns .true. if the timeout was reached, .false. otherwise. If timeoutFlag was *not* provided, a timeout condition will lead to a return code of rc \= ESMF_SUCCESS. Otherwise the return value of timeoutFlag is the sole indicator of a timeout condition.

[userRc] Return code set by userRoutine before returning.

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

19 SciComp Class

19.1 Description

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

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

19.2 Use and Examples

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

19.2.1 Use ESMF_SciComp and Attach Attributes

This example illustrates the use of the ESMF_SciComp to attach Attributes within a Component hierarchy. The hierarchy includes Coupler, Gridded, and Science Components and Attributes are attached to the Science Components. For demonstrable purposes, we'll add some CIM Component attributes to the Gridded Component.

Create the top 2 levels of the Component hierarchy. This example creates a parent Coupler Component and 2 Gridded Components as children.

```
! Create top-level Coupler Component
cplcomp = ESMF_CplCompCreate(name="coupler_component", rc=rc)

! Create Gridded Component for Atmosphere
atmcomp = ESMF_GridCompCreate(name="Atmosphere", rc=rc)

! Create Gridded Component for Ocean
ocncomp = ESMF_GridCompCreate(name="Ocean", rc=rc)
```

Now add CIM Attribute packages to the Component. Also, add a CIM Component Properties package, to contain two custom attributes.

```
convCIM = 'CIM 1.5'
purpComp = 'ModelComp'
purpProp = 'CompProp'
purpField = 'Inputs'
purpPlatform = 'Platform'

convISO = 'ISO 19115'
purpRP = 'RespParty'
purpCitation = 'Citation'

! Add CIM Attribute package to the Science Component
call ESMF_AttributeAdd(atmcomp, convention=convCIM, &
  purpose=purpComp, attpack=attpack, rc=rc)
```

The Attribute package can also be retrieved in a multi-Component setting like this:

```
call ESMF_AttributeGetAttPack(atmcomp, convCIM, purpComp, &
                              attpack=attpack, rc=rc)
```

Now, add some CIM Component attributes to the Atmosphere Grid Component.

```
!
! Top-level model component attributes, set on gridded component
!
call ESMF_AttributeSet(atmcomp, 'ShortName', 'EarthSys_Atmos', &
  attpack=attpack, rc=rc)
```

```
call ESMF_AttributeSet(atmcomp, 'LongName', &
  'Earth System High Resolution Global Atmosphere Model', &
  attpack=attpack, rc=rc)



call ESMF_AttributeSet(atmcomp, 'Description', &
  'EarthSys brings together expertise from the global ' // &
  'community in a concerted effort to develop coupled ' // &
  'climate models with increased horizontal resolutions.  ' // &
  'Increasing the horizontal resolution of coupled climate ' // &
  'models will allow us to capture climate processes and ' // &
  'weather systems in much greater detail.', &
  attpack=attpack, rc=rc)



call ESMF_AttributeSet(atmcomp, 'Version', '2.0', &
  attpack=attpack, rc=rc)



call ESMF_AttributeSet(atmcomp, 'ReleaseDate', '2009-01-01T00:00:00Z', &
  attpack=attpack, rc=rc)



call ESMF_AttributeSet(atmcomp, 'ModelType', 'aerosol', &
  attpack=attpack, rc=rc)



call ESMF_AttributeSet(atmcomp, 'URL', &
  'www.earthsys.org', attpack=attpack, rc=rc)
```

Now create a set of Science Components as a children of the Atmosphere Gridded Component. The hierarchy is as follows:

- Atmosphere
  - AtmosDynamicalCore
    - AtmosAdvection
  - AtmosRadiation

After each Component is created, we need to link it with its parent Component. We then add some standard CIM Component properties as well as Scientific Properties to each of these components.

```
!
! Atmosphere Dynamical Core Science Component
!
dc_scicomp = ESMF_SciCompCreate(name="AtmosDynamicalCore", rc=rc)
```

```
call ESMF_AttributeAdd(dc_scicomp,  &
                       convention=convCIM, purpose=purpComp, &
                       attpack=attpack, rc=rc)

call ESMF_AttributeSet(dc_scicomp, "ShortName", "AtmosDynamicalCore", &
                       attpack=attpack, rc=rc)
call ESMF_AttributeSet(dc_scicomp, "LongName", &
                       "Atmosphere Dynamical Core", &
                       attpack=attpack, rc=rc)


purpSci = 'SciProp'

dc_sciPropAtt(1) = 'TopBoundaryCondition'
dc_sciPropAtt(2) = 'HeatTreatmentAtTop'
dc_sciPropAtt(3) = 'WindTreatmentAtTop'

call ESMF_AttributeAdd(dc_scicomp,  &
                       convention=convCIM, purpose=purpSci, &
                       attrList=dc_sciPropAtt, &
                       attpack=attpack, rc=rc)

call ESMF_AttributeSet(dc_scicomp, 'TopBoundaryCondition', &
                       'radiation boundary condition', &
                       attpack=attpack, rc=rc)
call ESMF_AttributeSet(dc_scicomp, 'HeatTreatmentAtTop', &
                       'some heat treatment', &
                       attpack=attpack, rc=rc)
call ESMF_AttributeSet(dc_scicomp, 'WindTreatmentAtTop', &
                       'some wind treatment', &
                       attpack=attpack, rc=rc)



!
! Atmosphere Advection Science Component
!
adv_scicomp = ESMF_SciCompCreate(name="AtmosAdvection", rc=rc)


call ESMF_AttributeAdd(adv_scicomp,  &
                       convention=convCIM, purpose=purpComp, &
                       attpack=attpack, rc=rc)

call ESMF_AttributeSet(adv_scicomp, "ShortName", "AtmosAdvection", &
                       attpack=attpack, rc=rc)
call ESMF_AttributeSet(adv_scicomp, "LongName", "Atmosphere Advection", &
                       attpack=attpack, rc=rc)


adv_sciPropAtt(1) = 'TracersSchemeName'
adv_sciPropAtt(2) = 'TracersSchemeCharacteristics'
adv_sciPropAtt(3) = 'MomentumSchemeName'

call ESMF_AttributeAdd(adv_scicomp,  &
```

```
                        convention=convCIM, purpose=purpSci, &
                        attrList=adv_sciPropAtt, &
                        attpack=attpack, rc=rc)

    call ESMF_AttributeSet(adv_scicomp, 'TracersSchemeName', 'Prather', &
                        attpack=attpack, rc=rc)
    call ESMF_AttributeSet(adv_scicomp, 'TracersSchemeCharacteristics', &
                        'modified Euler', &
                        attpack=attpack, rc=rc)
    call ESMF_AttributeSet(adv_scicomp, 'MomentumSchemeName', 'Van Leer', &
                        attpack=attpack, rc=rc)



    !
    ! Atmosphere Radiation Science Component
    !
    rad_scicomp = ESMF_SciCompCreate(name="AtmosRadiation", rc=rc)


    call ESMF_AttributeAdd(rad_scicomp,  &
                        convention=convCIM, purpose=purpComp, &
                        attpack=attpack, rc=rc)

    call ESMF_AttributeSet(rad_scicomp, "ShortName", "AtmosRadiation", &
                        attpack=attpack, rc=rc)
    call ESMF_AttributeSet(rad_scicomp, "LongName", &
                        "Atmosphere Radiation", &
                        attpack=attpack, rc=rc)



    rad_sciPropAtt(1) = 'LongwaveSchemeType'
    rad_sciPropAtt(2) = 'LongwaveSchemeMethod'

    call ESMF_AttributeAdd(rad_scicomp,  &
                        convention=convCIM, purpose=purpSci, &
                        attrList=rad_sciPropAtt, &
                        attpack=attpack, rc=rc)

    call ESMF_AttributeSet(rad_scicomp, &
                        'LongwaveSchemeType', &
                        'wide-band model', &
                        attpack=attpack, rc=rc)
    call ESMF_AttributeSet(rad_scicomp, &
                        'LongwaveSchemeMethod', &
                        'two-stream', &
                        attpack=attpack, rc=rc)
```

Finally, destroy all of the Components.

```
  call ESMF_SciCompDestroy(rad_scicomp, rc=rc)
  call ESMF_SciCompDestroy(adv_scicomp, rc=rc)
  call ESMF_SciCompDestroy(dc_scicomp, rc=rc)
```

```
call ESMF_GridCompDestroy(atmcomp, rc=rc)
call ESMF_GridCompDestroy(ocncomp, rc=rc)
call ESMF_CplCompDestroy(cplcomp, rc=rc)
```

19.3 Restrictions and Future Work

1. **None.**

19.4 Class API

19.4.1 ESMF_SciCompAssignment(=) - SciComp assignment

INTERFACE:

```
interface assignment(=)
scicomp1 = scicomp2
```

ARGUMENTS:

```
type(ESMF_SciComp) :: scicomp1
type(ESMF_SciComp) :: scicomp2
```

DESCRIPTION:

Assign scicomp1 as an alias to the same ESMF SciComp object in memory as scicomp2. If scicomp2 is invalid, then scicomp1 will be equally invalid after the assignment.

The arguments are:

scicomp1 The `ESMF_SciComp` object on the left hand side of the assignment.

scicomp2 The `ESMF_SciComp` object on the right hand side of the assignment.

19.4.2 ESMF_SciCompOperator(==) - SciComp equality operator

INTERFACE:

```
interface operator(==)
  if (scicomp1 == scicomp2) then ... endif
            OR
  result = (scicomp1 == scicomp2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_SciComp), intent(in) :: scicomp1
type(ESMF_SciComp), intent(in) :: scicomp2
```

DESCRIPTION:

Test whether scicomp1 and scicomp2 are valid aliases to the same ESMF SciComp object in memory. For a more general comparison of two ESMF SciComps, going beyond the simple alias test, the ESMF_SciCompMatch() function (not yet implemented) must be used.

The arguments are:

scicomp1 The `ESMF_SciComp` object on the left hand side of the equality operation.

scicomp2 The `ESMF_SciComp` object on the right hand side of the equality operation.

19.4.3 ESMF_SciCompOperator(/=) - SciComp not equal operator

INTERFACE:

```
interface operator(/=)
  if (scicomp1 /= scicomp2) then ... endif
            OR
  result = (scicomp1 /= scicomp2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_SciComp), intent(in) :: scicomp1
type(ESMF_SciComp), intent(in) :: scicomp2
```

DESCRIPTION:

Test whether scicomp1 and scicomp2 are *not* valid aliases to the same ESMF SciComp object in memory. For a more general comparison of two ESMF SciComps, going beyond the simple alias test, the ESMF_SciCompMatch() function (not yet implemented) must be used.

The arguments are:

scicomp1 The `ESMF_SciComp` object on the left hand side of the non-equality operation.

scicomp2 The `ESMF_SciComp` object on the right hand side of the non-equality operation.

19.4.4 ESMF_SciCompCreate - Create a SciComp

INTERFACE:

```
recursive function ESMF_SciCompCreate(name, rc)
```

RETURN VALUE:

```
type(ESMF_SciComp) :: ESMF_SciCompCreate
```

ARGUMENTS:

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

DESCRIPTION:

This interface creates an ESMF_SciComp object. The return value is the new ESMF_SciComp.

The arguments are:

[name] Name of the newly-created ESMF_SciComp. This name can be altered from within the ESMF_SciComp code once the initialization routine is called.

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

19.4.5 ESMF_SciCompDestroy - Release resources associated with a SciComp

INTERFACE:

```
subroutine ESMF_SciCompDestroy(scicomp, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Destroys an ESMF_SciComp, releasing the resources associated with the object.

The arguments are:

scicomp Release all resources associated with this ESMF_SciComp and mark the object as invalid. It is an error to pass this object into any other routines after being destroyed.

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

19.4.6 ESMF_SciCompGet - Get SciComp information

INTERFACE:

```
   subroutine ESMF_SciCompGet(scicomp, name, rc)
```

ARGUMENTS:

```
     type(ESMF_SciComp),       intent(in)            :: scicomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     character(len=*),         intent(out), optional :: name
     integer,                  intent(out), optional :: rc
```

DESCRIPTION:

Get information about an `ESMF_SciComp` object.

The arguments are:

scicomp The `ESMF_SciComp` object being queried.

[name] Return the name of the `ESMF_SciComp`.

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

19.4.7 ESMF_SciCompIsCreated - Check whether a SciComp object has been created

INTERFACE:

```
   function ESMF_SciCompIsCreated(scicomp, rc)
```

RETURN VALUE:

```
     logical :: ESMF_SciCompIsCreated
```

ARGUMENTS:

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

DESCRIPTION:

Return `.true.` if the `scicomp` has been created. Otherwise return `.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:

scicomp ESMF_SciComp queried.

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

19.4.8 ESMF_SciCompPrint - Print SciComp information

INTERFACE:

```
   subroutine ESMF_SciCompPrint(scicomp, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Prints information about an ESMF_SciComp to stdout.

The arguments are:

scicomp ESMF_SciComp to print.

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

19.4.9 ESMF_SciCompSet - Set or reset information about the SciComp

INTERFACE:

```
   subroutine ESMF_SciCompSet(scicomp, name, rc)
```

ARGUMENTS:

```
     type(ESMF_SciComp), intent(inout)          :: scicomp
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     character(len=*),    intent(in),  optional :: name
     integer,             intent(out), optional :: rc
```

DESCRIPTION:

Sets or resets information about an ESMF_SciComp.

The arguments are:

scicomp `ESMF_SciComp` to change.

[name] Set the name of the `ESMF_SciComp`.

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

19.4.10 ESMF_SciCompValidate - Check validity of a SciComp

INTERFACE:

```
   subroutine ESMF_SciCompValidate(scicomp, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Currently all this method does is to check that the `scicomp` was created.

The arguments are:

scicomp `ESMF_SciComp` to validate.

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

20 Fault-tolerant Component Tunnel

20.1 Description

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

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

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

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

20.2 Use and Examples

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

20.2.1 Creating an *actual* Component

The creation process of an *actual* Gridded Component, which will become one of the two end points of a Component Tunnel, is identical to the creation of a regular Gridded Component. On the actual side, an actual Component is very similar to a regular Component. Here the actual Component is created with a custom `petList`.

```
petList = (/0,1,2/)
actualComp = ESMF_GridCompCreate(petList=petList, name="actual", rc=rc)
```

20.2.2 Creating a *dual* Component

The same way an actual Component appears as a regular Component in the context of the actual side application, a *dual* Component is created as a regular Component on the dual side. A dual Gridded Component with custom `petList` is created using the regular create call.

```
petList = (/4,3,5/)
dualComp = ESMF_GridCompCreate(petList=petList, name="dual", rc=rc)
```

20.2.3 Setting up the *actual* side of a Component Tunnel

After creation, the regular procedure for registering the standard Component methods is followed for the actual Gridded Component.

```
call ESMF_GridCompSetServices(actualComp, userRoutine=setservices, &
  userRc=userRc, rc=rc)
```

So far the `actualComp` object is no different from a regular Gridded Component. In order to turn it into the *actual* end point of a Component Tunnel the `ServiceLoop()` method is called. Here the socket-based implementation is chosen.

```
call ESMF_GridCompServiceLoop(actualComp, port=61010, timeout=20, rc=rc)
```

This call opens the actual side of the Component Tunnel in form of a socket-based server, listening on `port` 61010. The `timeout` argument specifies how long the actual side will wait for the dual side to connect, before the actual side returns with a time out condition. The time out is set to 20 seconds.

At this point, before a dual Component connects to the other side of the Component Tunnel, it is possible to manually connect to the waiting actual Component. This can be useful when debugging connection issues. A convenient tool for this is the standard `telnet` application. Below is a transcript of such a connection. The manually typed commands are separate from the previous responses by a blank line.

```
$ telnet localhost 61010
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello from ESMF Actual Component server!

date
Tue Apr  3 21:53:03 2012

version
ESMF_VERSION_STRING: 5.3.0
```

If at any point the `telnet` session is manually shut down, the `ServiceLoop()` on the actual side will return with an error condition. The clean way to disconnect the `telnet` session, and to have the `ServiceLoop()` wait for a new connection, e.g. from a dual Component, is to send the `reconnect` command. This will automatically shut down the `telnet` connection.

```
reconnect
Actual Component server will reconnect now!
Connection closed by foreign host.
$
```

At this point the actual Component is back in listening mode, with a time out of 20 seconds, as specified during the ServiceLoop() call.

Before moving on to the dual side of the GridComp based Component Tunnel example, it should be pointed out that the exact same procedure is used to set up the actual side of a *CplComp* based Component Tunnel. Assuming that `actualCplComp` is a CplComp object for which SetServices has already been called, the actual side uses `ESMF_CplCompServiceLoop()` to start listening for connections from the dual side.

```
call ESMF_CplCompServiceLoop(actualCplComp, port=61011, timeout=2, &
  timeoutFlag=timeoutFlag, rc=rc)
```

Here the `timeoutFlag` is specified in order to prevent the expected time-out condition to be indicated through the return code. Instead, when `timeoutFlag` is present, the return code is still `ESMF_SUCCESS`, but `timeoutFlag` is set to `.true.` when a time-out occurs.

20.2.4 Setting up the *dual* side of a Component Tunnel

On the dual side, the `dualComp` object needs to be connected to the actual Component in order to complete the Component Tunnel. Instead of registering standard Component methods locally, a special variant of the `SetServices()` call is used to connect to the actual Component.

```
call ESMF_GridCompSetServices(dualComp, port=61010, server="localhost", &
  timeout=10, timeoutFlag=timeoutFlag, rc=rc)
```

The `port` and `server` arguments are used to connect to the desired actual Component. The time out of 10 seconds ensures that if the actual Component is not available, a time out condition is returned instead of resulting in a hang. The `timeoutFlag` argument further absorbs the time out condition, either returning as `.true.` or `.false.`. In this mode the standard `rc` will indicate success even when a time out condition was reached.

20.2.5 Invoking standard Component methods through a Component Tunnel

Once a Component Tunnel is established, the actual Component is fully under the control of the dual Component. A standard Component method invoked on the dual Component is not executed by the dual Component itself, but by the actual Component instead. In fact, it is the entry points registered with the actual Component that are executed when standard methods are invoked on the dual Component. The connected `dualComp` object serves as a portal through which the connected `actualComp` becomes accessible on the dual side.

Typically the first standard method called is the `CompInitialize()` routine.

```
call ESMF_GridCompInitialize(dualComp, timeout=10, timeoutFlag=timeoutFlag, &
  userRc=userRc, rc=rc)
```

Again, the `timeout` argument serves to prevent the dual side from hanging if the actual Component application has experienced a catastrophic condition and is no longer available, or takes longer than expected. The presence of the `timeoutFlag` allows time out conditions to be caught gracefully, so the dual side can deal with it in an orderly fashion, instead of triggering an application abort due to an error condition.

The `CompRun()` and `CompFinalize()` methods follow the same format.

```
call ESMF_GridCompRun(dualComp, timeout=10, timeoutFlag=timeoutFlag, &
  userRc=userRc, rc=rc)
```

```
call ESMF_GridCompFinalize(dualComp, timeout=10, timeoutFlag=timeoutFlag, &
  userRc=userRc, rc=rc)
```

20.2.6 The non-blocking option to invoke standard Component methods through a Component Tunnel

Standard Component methods called on a connected dual Component are executed on the actual side, across the PETs of the actual Component. By default the dual Component PETs are blocked until the actual Component has finished executing the invoked Component method, or until a time out condition has been reached. In many practical applications a more loose synchronization between dual and actual Components is useful. Having the PETs of a dual Component return immediately from a standard Component method allows multiple dual Component, on the same PETs, to control multiple actual Components. If the actual Components are executing in separate executables, or the same executable but on exclusive sets of PETs, they can execute concurrently, even with the controlling dual Components all running on the same PETs. The non-blocking dual side regains control over the actual Component by synchronizing through the CompWait() call.

Any of the standard Component methods can be called in non-blocking mode by setting the optional `syncflag` argument to `ESMF_SYNC_NONBLOCKING`.

```
call ESMF_GridCompInitialize(dualComp, syncflag=ESMF_SYNC_NONBLOCKING, rc=rc)
```

If communication between the dual and the actual Component was successful, this call will return immediately on all of the dual Component PETs, while the actual Component continues to execute the invoked Component method. However, if the dual Component has difficulties reaching the actual Component, the call will block on all dual PETs until successful contact was made, or the default time out (3600 seconds, i.e. 1 hour) has been reached. In most cases a shorter time out condition is desired with the non-blocking option, as shown below.

First the dual Component must wait for the outstanding method.

```
call ESMF_GridCompWait(dualComp, rc=rc)
```

Now the same non-blocking CompInitialize() call is issued again, but this time with an explicit 10 second time out.

```
call ESMF_GridCompInitialize(dualComp, syncflag=ESMF_SYNC_NONBLOCKING, &
  timeout=10, timeoutFlag=timeoutFlag, rc=rc)
```

This call is guaranteed to return within 10 seconds, or less, on the dual Component PETs, either without time out condition, indicating that the actual Component has been contacted successfully, or with time out condition, indicating that the actual Component was unreachable at the time. Either way, the dual Component PETs are back under user control quickly.

Calling the CompWait() method on the dual Component causes the dual Component PETs to block until the actual Component method has returned, or a time out condition has been reached.

```
call ESMF_GridCompWait(dualComp, userRc=userRc, rc=rc)
```

The default time out for CompWait() is 3600 seconds, i.e. 1 hour, just like for the other Component methods. However, the semantics of a time out condition under CompWait() is different from the other Component methods. Typically the `timeout` is simply the maximum time that any communication between dual and actual Component is allowed to take before a time out condition is raised. For CompWait(), the `timeout` is the maximum time that an actual Component is allowed to execute before reporting back to the dual Component. Here, even with the default time out, the dual Component would return from CompWait() immediately with a time out condition if the actual Component has already been executing for over 1 hour, and is not already waiting to report back when the dual Component calls CompWait(). On the other hand, if it has only been 30 minutes since CompInitialize() was called on the dual Component, then the actual Component still has 30 minutes before CompWait() returns with a time out condition. During this time (or until the actual Component returns) the dual Component PETs are blocked.

A standard Component method is invoked in non-blocking mode.

```
call ESMF_GridCompRun(dualComp, syncflag=ESMF_SYNC_NONBLOCKING, &
  timeout=10, timeoutFlag=timeoutFlag, rc=rc)
```

Once the user code on the dual side is ready to regain control over the actual Component it calls CompWait() on the dual Component. Here a `timeout` of 60s is specified, meaning that the total execution time the actual Component spends in the registered Run() routine may not exceed 60s before CompWait() returns with a time out condition.

```
call ESMF_GridCompWait(dualComp, timeout=60, userRc=userRc, rc=rc)
```

20.2.7 Destroying a connected *dual* Component

A dual Component that is connected to an actual Component through a Component Tunnel is destroyed the same way a regular Component is. The only difference is that a connected dual Component may specify a `timeout` argument to the `CompDestroy()` call.

```
call ESMF_GridCompDestroy(dualComp, timeout=10, rc=rc)
```

The `timeout` argument again ensures that the dual side does not hang indefinitely in case the actual Component has become unavailable. If the actual Component is available, the destroy call will indicate to the actual Component that it should break out of the `ServiceLoop()`. Either way, the local dual Component is destroyed.

20.2.8 Destroying a connected *actual* Component

An actual Component that is in a `ServiceLoop()` must first return from that call before it can be destroyed. This can either happen when a connected dual Component calls its `CompDestroy()` method, or if the `ServiceLoop()` reaches the specified time out condition. Either way, once control has been returned to the user code, the actual Component is destroyed in the same way a regular Component is, by calling the destroy method.

```
call ESMF_GridCompDestroy(actualComp, rc=rc)
```

20.3 Restrictions and Future Work

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

21 State Class

21.1 Description

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

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

States can contain Arrays, ArrayBundles, Fields, FieldBundles, and other States. They cannot directly contain native language arrays (i.e. Fortran or C style arrays). Objects in a State must span the VM on which they are running. For sequentially executing components which run on the same set of PETs this happens by calling the object create methods on each PET, creating the object in unison. For concurrently executing components which are running on

subsets of PETs, an additional method, called ESMF_StateReconcile(), is provided by ESMF to broadcast information about objects which were created in sub-components.

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

21.2 Constants

21.2.1 ESMF_STATEINTENT

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

The type of this flag is:

type(ESMF_StateIntent_Flag)

The valid values are:

ESMF_STATEINTENT_IMPORT Contains data to be imported into a component.

ESMF_STATEINTENT_EXPORT Contains data to be exported out of a component.

ESMF_STATEINTENT_UNSPECIFIED The intent has not been specified.

21.2.2 ESMF_STATEITEM

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

The type of this flag is:

type(ESMF_StateItem_Flag)

The valid values are:

ESMF_STATEITEM_ARRAY Refers to an ESMF_Array within an ESMF_State.

ESMF_STATEITEM_ARRAYBUNDLE Refers to an ESMF_Array within an ESMF_State.

ESMF_STATEITEM_FIELD Refers to a ESMF_Field within an ESMF_State.

ESMF_STATEITEM_FIELDBUNDLE Refers to a ESMF_FieldBundle within an ESMF_State.

ESMF_STATEITEM_ROUTEHANDLE Refers to a ESMF_RouteHandle within an ESMF_State.

ESMF_STATEITEM_STATE Refers to a ESMF_State within an ESMF_State.

21.3 Use and Examples

A Gridded Component generally has one associated import State and one export State. Generally the States associated with a Gridded Component will be created by the Gridded Component's parent component. In many cases, the States will be created containing no data. Both the empty States and the newly created Gridded Component are passed by

the parent component into the Gridded Component's initialize method. This is where the States get prepared for use and the import State is first filled with data.

States can be filled with data items that do not yet have data allocated. Fields, FieldBundles, Arrays, and ArrayBundles each have methods that support their creation without actual data allocation - the Grid and Attributes are set up but no Fortran array of data values is allocated. In this approach, when a State is passed into its associated Gridded Component's initialize method, the incomplete Arrays, Fields, FieldBundles, and ArrayBundles within the State can allocate or reference data inside the initialize method.

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

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

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

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

21.3.1 State create and destroy

States can be created and destroyed at any time during application execution. The `ESMF_StateCreate()` routine can take many different combinations of optional arguments. Refer to the API description for all possible methods of creating a State. An empty State can be created by providing only a name and type for the intended State:

```
state = ESMF_StateCreate(name, stateintent=ESMF_STATEINTENT_IMPORT, rc=rc)
```

When finished with an `ESMF_State`, the `ESMF_StateDestroy` method removes it. However, the objects inside the `ESMF_State` created externally should be destroyed separately, since objects can be added to more than one `ESMF_State`.

21.3.2 Add items to a State

Creation of an empty ESMF_State, and adding an ESMF_FieldBundle to it. Note that the ESMF_FieldBundle does not get destroyed when the ESMF_State is destroyed; the ESMF_State only contains a reference to the objects it contains. It also does not make a copy; the original objects can be updated and code accessing them by using the ESMF_State will see the updated version.

```
statename = "Ocean"
state2 = ESMF_StateCreate(name=statename,  &
                          stateintent=ESMF_STATEINTENT_EXPORT, rc=rc)


bundlename = "Temperature"
bundle1 = ESMF_FieldBundleCreate(name=bundlename, rc=rc)
print *, "FieldBundle Create returned", rc


call ESMF_StateAdd(state2, (/bundle1/), rc=rc)
print *, "StateAdd returned", rc


call ESMF_StateDestroy(state2, rc=rc)


call ESMF_FieldBundleDestroy(bundle1, rc=rc)
```

21.3.3 Add placeholders to a State

If a component could potentially produce a large number of optional items, one strategy is to add the names only of those objects to the ESMF_State. Other components can call framework routines to set the ESMF_NEEDED flag to indicate they require that data. The original component can query this flag and then produce only the data that is required by another component.

```
statename = "Ocean"
state3 = ESMF_StateCreate(name=statename,  &
                          stateintent=ESMF_STATEINTENT_EXPORT, rc=rc)


dataname = "Downward wind:needed"
call ESMF_AttributeSet (state3, dataname, .false., rc=rc)


dataname = "Humidity:needed"
call ESMF_AttributeSet (state3, dataname, .false., rc=rc)
```

21.3.4 Mark an item NEEDED

How to set the NEEDED state of an item.

```
    dataname = "Downward wind:needed"
    call ESMF_AttributeSet (state3, name=dataname, value=.true., rc=rc)
```

21.3.5 Create a NEEDED item

Query an item for the NEEDED status, and creating an item on demand. Similar flags exist for "Ready", "Valid", and "Required for Restart", to mark each data item as ready, having been validated, or needed if the application is to be checkpointed and restarted. The flags are supported to help coordinate the data exchange between components.

```
    dataname = "Downward wind:needed"
    call ESMF_AttributeGet (state3, dataname, value=neededFlag, rc=rc)


    if (rc == ESMF_SUCCESS .and. neededFlag) then
        bundlename = dataname
        bundle2 = ESMF_FieldBundleCreate(name=bundlename, rc=rc)


        call ESMF_StateAdd(state3, (/bundle2/), rc=rc)


    else
        print *, "Data not marked as needed", trim(dataname)
    endif
```

21.3.6 ESMF_StateReconcile() usage

The set services routines are used to tell ESMF which routine hold the user code for the initialize, run, and finalize blocks of user level Components. These are the separate subroutines called by the code below.

```
! Initialize routine which creates "field1" on PETs 0 and 1
subroutine comp1_init(gcomp, istate, ostate, clock, rc)
    type(ESMF_GridComp)  :: gcomp
    type(ESMF_State)     :: istate, ostate
    type(ESMF_Clock)     :: clock
    integer, intent(out) :: rc

    type(ESMF_Field) :: field1
    integer :: localrc

    print *, "i am comp1_init"

    field1 = ESMF_FieldEmptyCreate(name="Comp1 Field", rc=localrc)

    call ESMF_StateAdd(istate, (/field1/), rc=localrc)

    rc = localrc
```

```
end subroutine comp1_init

! Initialize routine which creates "field2" on PETs 2 and 3
subroutine comp2_init(gcomp, istate, ostate, clock, rc)
    type(ESMF_GridComp)  :: gcomp
    type(ESMF_State)     :: istate, ostate
    type(ESMF_Clock)     :: clock
    integer, intent(out) :: rc

    type(ESMF_Field) :: field2
    integer :: localrc

    print *, "i am comp2_init"

    field2 = ESMF_FieldEmptyCreate(name="Comp2 Field", rc=localrc)

    call ESMF_StateAdd(istate, (/field2/), rc=localrc)

    rc = localrc

end subroutine comp2_init

subroutine comp_dummy(gcomp, rc)
   type(ESMF_GridComp)  :: gcomp
   integer, intent(out) :: rc

   rc = ESMF_SUCCESS
end subroutine comp_dummy


! !PROGRAM: ESMF_StateReconcileEx - State reconciliation
!
! !DESCRIPTION:
!
! This program shows examples of using the State Reconcile function
!-------------------------------------------------------------------------
#include "ESMF.h"

    ! ESMF Framework module
    use ESMF
    use ESMF_TestMod
    use ESMF_StateReconcileEx_Mod
    implicit none

    ! Local variables
    integer :: rc, petCount
    type(ESMF_State) :: state1
    type(ESMF_GridComp) :: comp1, comp2
    type(ESMF_VM) :: vm
    character(len=ESMF_MAXSTR) :: comp1name, comp2name, statename
```

A Component can be created which will run only on a subset of the current PET list.

```
! Get the global VM for this job.
call ESMF_VMGetGlobal(vm=vm, rc=rc)

comp1name = "Atmosphere"
comp1 = ESMF_GridCompCreate(name=comp1name, petList=(/ 0, 1 /), rc=rc)
print *, "GridComp Create returned, name = ", trim(comp1name)



comp2name = "Ocean"
comp2 = ESMF_GridCompCreate(name=comp2name, petList=(/ 2, 3 /), rc=rc)
print *, "GridComp Create returned, name = ", trim(comp2name)



statename = "Ocn2Atm"
state1 = ESMF_StateCreate(name=statename, rc=rc)
```

Here we register the subroutines which should be called for initialization. Then we call ESMF_GridCompInitialize() on all PETs, but the code runs only on the PETs given in the petList when the Component was created.

Because this example is so short, we call the entry point code directly instead of the normal procedure of nesting it in a separate SetServices() subroutine.

```
! This is where the VM for each component is initialized.
! Normally you would call SetEntryPoint inside set services,
! but to make this example very short, they are called inline below.
! This is o.k. because the SetServices routine must execute from within
! the parent component VM.
call ESMF_GridCompSetVM(comp1, comp_dummy, rc=rc)



call ESMF_GridCompSetVM(comp2, comp_dummy, rc=rc)



call ESMF_GridCompSetServices(comp1, userRoutine=comp_dummy, rc=rc)



call ESMF_GridCompSetServices(comp2, userRoutine=comp_dummy, rc=rc)



print *, "ready to set entry point 1"
call ESMF_GridCompSetEntryPoint(comp1, ESMF_METHOD_INITIALIZE, &
     comp1_init, rc=rc)
```

```
    print *, "ready to set entry point 2"
    call ESMF_GridCompSetEntryPoint(comp2, ESMF_METHOD_INITIALIZE, &
         comp2_init, rc=rc)




    print *, "ready to call init for comp 1"
    call ESMF_GridCompInitialize(comp1, exportState=state1, rc=rc)



    print *, "ready to call init for comp 2"
    call ESMF_GridCompInitialize(comp2, exportState=state1, rc=rc)
```

Now we have `state1` containing `field1` on PETs 0 and 1, and `state1` containing `field2` on PETs 2 and 3. For the code to have a rational view of the data, we call `ESMF_StateReconcile` which determines which objects are missing from any PET, and communicates information about the object. After the call to reconcile, all `ESMF_State` objects now have a consistent view of the data.

```
    print *, "State before calling StateReconcile()"
    call ESMF_StatePrint(state1, rc=rc)




    call ESMF_StateReconcile(state1, vm=vm, rc=rc)




    print *, "State after calling StateReconcile()"
    call ESMF_StatePrint(state1, rc=rc)


end program ESMF_StateReconcileEx
```

21.3.7 Read Arrays from a NetCDF file and add to a State

This program shows an example of reading and writing Arrays from a State from/to a NetCDF file.

```
    ! ESMF Framework module
    use ESMF
    use ESMF_TestMod
    implicit none
```

```
! Local variables
type(ESMF_State) :: state
type(ESMF_Array) :: latArray, lonArray, timeArray, humidArray, &
                    tempArray, pArray, rhArray
type(ESMF_VM) :: vm
integer :: localPet, rc
```

The following line of code will read all Array data contained in a NetCDF file, place them in ESMF_Arrays and add them to an ESMF_State. Only PET 0 reads the file; the States in the other PETs remain empty. Currently, the data is not decomposed or distributed; each PET has only 1 DE and only PET 0 contains data after reading the file. Future versions of ESMF will support data decomposition and distribution upon reading a file.

Note that the third party NetCDF library must be installed. For more details, see the "ESMF Users Guide", "Building and Installing the ESMF, Third Party Libraries, NetCDF" and the website http://www.unidata.ucar.edu/software/netcdf.

```
! Read the NetCDF data file into Array objects in the State on PET 0
call ESMF_StateRead(state, "io_netcdf_testdata.nc", rc=rc)

! If the NetCDF library is not present (on PET 0), cleanup and exit
if (rc == ESMF_RC_LIB_NOT_PRESENT) then
  call ESMF_StateDestroy(state, rc=rc)
  goto 10
endif
```

Only reading data into ESMF_Arrays is supported at this time; ESMF_ArrayBundles, ESMF_Fields, and ESMF_FieldBundles will be supported in future releases of ESMF.

21.3.8 Print Array data from a State

To see that the State now contains the same data as in the file, the following shows how to print out what Arrays are contained within the State and to print the data contained within each Array. The NetCDF utility "ncdump" can be used to view the contents of the NetCDF file. In this example, only PET 0 will contain data.

```
if (localPet == 0) then
  ! Print the names and attributes of Array objects contained in the State
  call ESMF_StatePrint(state, rc=rc)

  ! Get each Array by name from the State
  call ESMF_StateGet(state, "lat",  latArray,   rc=rc)
  call ESMF_StateGet(state, "lon",  lonArray,   rc=rc)
  call ESMF_StateGet(state, "time", timeArray,  rc=rc)
  call ESMF_StateGet(state, "Q",    humidArray, rc=rc)
  call ESMF_StateGet(state, "TEMP", tempArray,  rc=rc)
  call ESMF_StateGet(state, "p",    pArray,     rc=rc)
  call ESMF_StateGet(state, "rh",   rhArray,    rc=rc)

  ! Print out the Array data
  call ESMF_ArrayPrint(latArray,   rc=rc)
  call ESMF_ArrayPrint(lonArray,   rc=rc)
```

```
  call ESMF_ArrayPrint(timeArray,  rc=rc)
  call ESMF_ArrayPrint(humidArray, rc=rc)
  call ESMF_ArrayPrint(tempArray,  rc=rc)
  call ESMF_ArrayPrint(pArray,     rc=rc)
  call ESMF_ArrayPrint(rhArray,    rc=rc)
endif
```

Note that the Arrays "lat", "lon", and "time" hold spatial and temporal coordinate data for the dimensions latitude, longitude and time, respectively. These will be used in future releases of ESMF to create `ESMF_Grids`.

21.3.9 Write Array data within a State to a NetCDF file

All the Array data within the State on PET 0 can be written out to a NetCDF file as follows:

```
! Write Arrays within the State on PET 0 to a NetCDF file
call ESMF_StateWrite(state, "io_netcdf_testdata_out.nc", rc=rc)
```

Currently writing is limited to PET 0; future versions of ESMF will allow parallel writing, as well as parallel reading.

21.4 Restrictions and Future Work

1. **No synchronization of object IDs at object create time.** Object IDs are used during the reconcile process to identify objects which are unknown to some subset of the PETs in the currently running VM. Object IDs are assigned in sequential order at object create time.

   One important request by the user community during the ESMF object design was that there be no communication overhead or synchronization when creating distributed ESMF objects. As a consequence it is required to create these objects in **unison** across all PETs in order to keep the ESMF object identification in sync.

21.5 Design and Implementation Notes

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

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

   When running components on subsets of the original VM all the PETs can create consistent objects but then when they are put into a State and passed to a component with a different VM and a different set of PETs, a communication call (reconcile) must be made to communicate the missing information to the PETs which were not involved in the original object creation. The reconcile call broadcasts object lists; those PETs which are

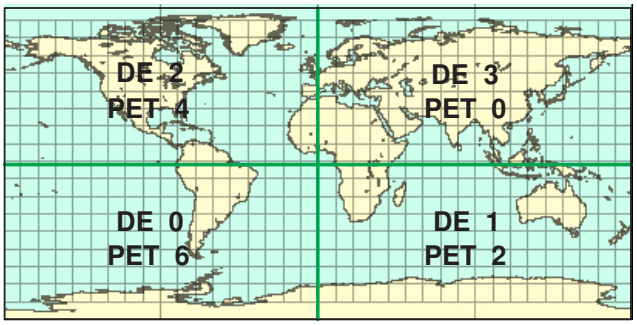


Figure 7: The mapping of PETs (processors) to DEs (data) in the source grid created by `user_model1.F90` in the FieldExcl system test.

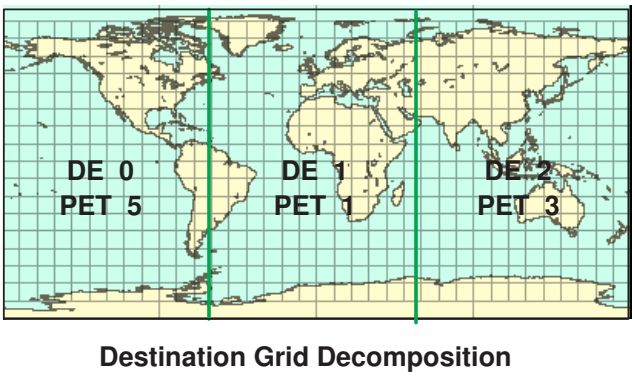


Figure 8: The mapping of PETs (processors) to DEs (data) in the destination grid created by `user_model2.F90` in the FieldExcl system test.

missing any objects in the total list can receive enough information to reconstruct a proxy object which contains all necessary information about that object, with no local data, on that PET. These proxy objects can be queried by ESMF routines to determine the amount of data and what PETs contain data which is destined to be moved to the local PET (for receiving data) and conversely, can determine which other PETs are going to receive data and how much (for sending data).

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

In the coupler code, all PETs must make the reconcile call before accessing data in the State. On PETs which already contain data, the objects are unchanged. On PETs which were not involved during the creation of the FieldBundles or Fields, the reconcile call adds an object to the State which contains all the same metadata

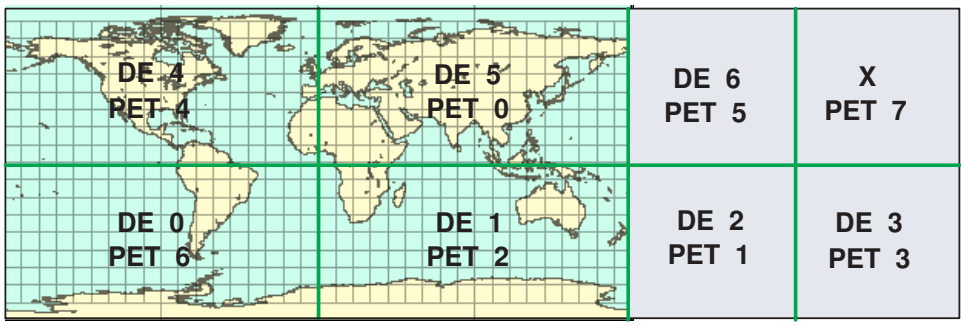


Figure 9: The mapping of PETs (processors) to DEs (data) in the source grid after the reconcile call in `user_coupler.F90` in the FieldExcl system test.

associated with the object, but creates a slightly different Grid object, called a Proxy Grid. These PETs contain no local data, so the Array object is empty, and the DELayout for the Grid is like this:

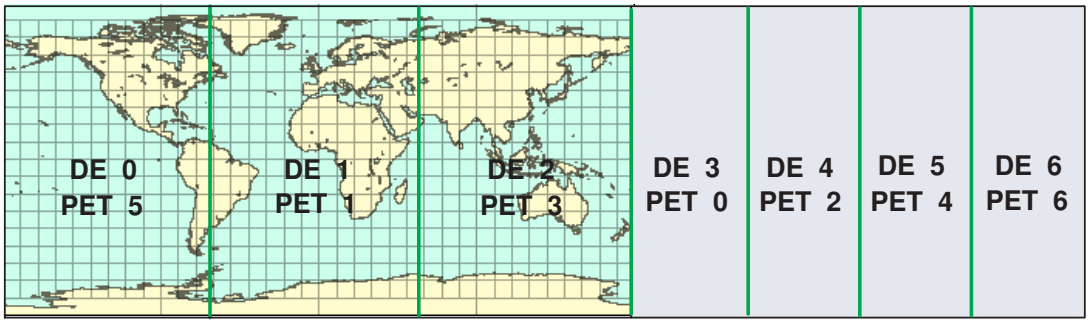
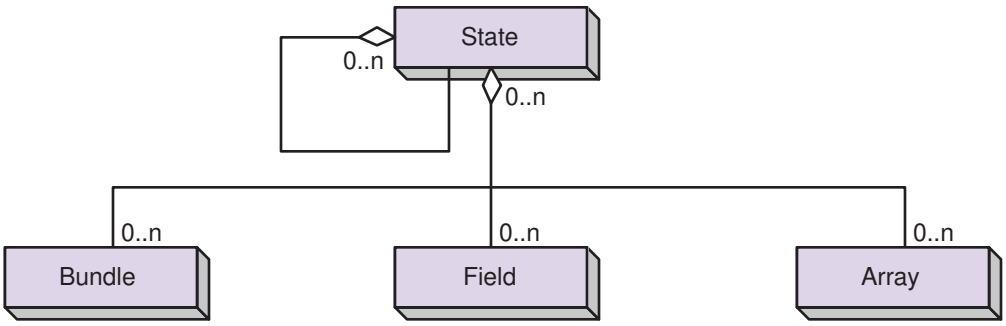


Figure 10: The mapping of PETs (processors) to DEs (data) in the destination grid after the reconcile call in `user_coupler.F90` in the FieldExcl system test.

21.6 Object Model

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



21.7 Class API

21.7.1 ESMF_StateAssignment(=) - State assignment

INTERFACE:

```
interface assignment(=)
state1 = state2
```

ARGUMENTS:

```
type(ESMF_State) :: state1
type(ESMF_State) :: state2
```

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:

Assign state1 as an alias to the same ESMF State object in memory as state2. If state2 is invalid, then state1 will be equally invalid after the assignment.

The arguments are:

state1 The `ESMF_State` object on the left hand side of the assignment.

state2 The `ESMF_State` object on the right hand side of the assignment.

21.7.2 ESMF_StateOperator(==) - State equality operator

INTERFACE:

```
interface operator(==)
if (state1 == state2) then ... endif
OR
result = (state1 == state2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state1
type(ESMF_State), intent(in) :: state2
```

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:

Test whether state1 and state2 are valid aliases to the same ESMF State object in memory. For a more general comparison of two ESMF States, going beyond the simple alias test, the ESMF_StateMatch() function (not yet implemented) must be used.

The arguments are:

state1 The ESMF_State object on the left hand side of the equality operation.

state2 The ESMF_State object on the right hand side of the equality operation.

21.7.3 ESMF_StateOperator(/=) - State not equal operator

INTERFACE:

```
interface operator(/=)
if (state1 /= state2) then ... endif
OR
result = (state1 /= state2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state1
type(ESMF_State), intent(in) :: state2
```

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:

Test whether state1 and state2 are *not* valid aliases to the same ESMF State object in memory. For a more general comparison of two ESMF States, going beyond the simple alias test, the ESMF_StateMatch() function (not yet implemented) must be used.

The arguments are:

state1 The ESMF_State object on the left hand side of the non-equality operation.

state2 The ESMF_State object on the right hand side of the non-equality operation.

21.7.4 ESMF_StateAdd - Add a list of items to a State

INTERFACE:

```
subroutine ESMF_StateAdd(state, <itemList>, relaxedFlag, rc)
```

ARGUMENTS:

```
   type(ESMF_State), intent(inout) :: state
   <itemList>, see below for supported values
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
   logical, intent(in), optional :: relaxedFlag
   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:

Add a list of items to a `ESMF_State`. It is an error if any item in <itemlist> already matches, by name, an item already contained in `state`.

Supported values for <itemList> are:

type(ESMF_Array), intent(in) :: arrayList(:)

type(ESMF_ArrayBundle), intent(in) :: arraybundleList(:)

type(ESMF_Field), intent(in) :: fieldList(:)

type(ESMF_FieldBundle), intent(in) :: fieldbundleList(:)

type(ESMF_RouteHandle), intent(in) :: routehandleList(:)

type(ESMF_State), intent(in) :: nestedStateList(:)

The arguments are:

state An `ESMF_State` to which the <itemList> will be added.

<itemList> The list of items to be added. This is a reference only; when the `ESMF_State` is destroyed the <itemList> items contained within it will not be destroyed. Also, the items in the <itemList> cannot be safely destroyed before the `ESMF_State` is destroyed. Since <itemList> items can be added to multiple containers, it remains the responsibility of the user to manage their destruction when they are no longer in use.

[relaxedflag] A setting of `.true.` indicates a relaxed definition of "add", where it is *not* an error if `<itemList>` contains items with names that are found in `state`. The `State` is left unchanged for these items. For `.false.` this is treated as an error condition. The default setting is `.false.`.

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

21.7.5 ESMF_StateAddReplace - Add or replace a list of items to a State

INTERFACE:

```
subroutine ESMF_StateAddReplace(state, <itemList>, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout) :: state
<itemList>, see below for supported values
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
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:

Add or replace a list of items to an `ESMF_State`. If an item in <itemList> does not match any items already present in `state`, it is added. Items with names already present in the `state` replace the existing item.

Supported values for <itemList> are:

type(ESMF_Array), intent(in) :: arrayList(:)

type(ESMF_ArrayBundle), intent(in) :: arraybundleList(:)

type(ESMF_Field), intent(in) :: fieldList(:)

type(ESMF_FieldBundle), intent(in) :: fieldbundleList(:)

type(ESMF_RouteHandle), intent(in) :: routehandleList(:)

type(ESMF_State), intent(in) :: nestedStateList(:)

The arguments are:

state An `ESMF_State` to which the <itemList> will be added or replaced.

<itemList> The list of items to be added or replaced. This is a reference only; when the `ESMF_State` is destroyed the <itemList> items contained within it will not be destroyed. Also, the items in the <itemList> cannot be safely destroyed before the `ESMF_State` is destroyed. Since <itemList> items can be added to multiple containers, it remains the responsibility of the user to manage their destruction when they are no longer in use.

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

21.7.6 ESMF_StateCreate - Create a new State

INTERFACE:

```
function ESMF_StateCreate(stateintent, &
             arrayList, arraybundleList, &
             fieldList, fieldbundleList, &
             nestedStateList, &
             routehandleList, name, vm, rc)
```

RETURN VALUE:

```
type(ESMF_State) :: ESMF_StateCreate
```

ARGUMENTS:

```
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_StateIntent_Flag), intent(in), optional :: stateintent
    type(ESMF_Array), intent(in), optional :: arrayList(:)
    type(ESMF_ArrayBundle), intent(in), optional :: arraybundleList(:)
    type(ESMF_Field), intent(in), optional :: fieldList(:)
    type(ESMF_FieldBundle), intent(in), optional :: fieldbundleList(:)
    type(ESMF_State), intent(in), optional :: nestedStateList(:)
    type(ESMF_RouteHandle), intent(in), optional :: routehandleList(:)
    character(len=*), intent(in), optional :: name
    type(ESMF_VM), intent(in), 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:

  8.1.0 Added argument `vm` to support object creation on a different VM than that of the current context.

DESCRIPTION:

Create a new `ESMF_State`, set default characteristics for objects added to it, and optionally add initial objects to it.

The arguments are:

[stateintent] Import or Export `ESMF_State`. Valid values are `ESMF_STATEINTENT_IMPORT`, `ESMF_STATEINTENT_EXPORT`, or `ESMF_STATEINTENT_UNSPECIFIED` The default is `ESMF_STATEINTENT_UNSPECIFIED`.

[arrayList] A list (Fortran array) of `ESMF_Array`s.

[arraybundleList] A list (Fortran array) of `ESMF_ArrayBundle`s.

[fieldList] A list (Fortran array) of `ESMF_Field`s.

[fieldbundleList] A list (Fortran array) of `ESMF_FieldBundle`s.

[nestedStateList] A list (Fortran array) of `ESMF_State`s to be nested inside the outer `ESMF_State`.

[routehandleList] A list (Fortran array) of `ESMF_RouteHandle`s.

[name] Name of this `ESMF_State` object. A default name will be generated if none is specified.

[vm] If present, the State object is created on the specified `ESMF_VM` object. The default is to create on the VM of the current component context.

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

21.7.7 ESMF_StateDestroy - Release resources for a State

INTERFACE:

```
recursive subroutine ESMF_StateDestroy(state, noGarbage, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout) :: state
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
logical, intent(in), optional :: noGarbage
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:

  8.1.0 Added argument noGarbage. The argument provides a mechanism to override the default garbage collection mechanism when destroying an ESMF object.

DESCRIPTION:

Releases resources associated with this ESMF_State. Actual objects added to ESMF_States will not be destroyed, it remains the responsibility of the user to destroy these objects in the correct context.

The arguments are:

state Destroy contents of this ESMF_State.

[noGarbage] If set to .TRUE. the object will be fully destroyed and removed from the ESMF garbage collection system. Note however that under this condition ESMF cannot protect against accessing the destroyed object through dangling aliases – a situation which may lead to hard to debug application crashes.

It is generally recommended to leave the noGarbage argument set to .FALSE. (the default), and to take advantage of the ESMF garbage collection system which will prevent problems with dangling aliases or incorrect sequences of destroy calls. However this level of support requires that a small remnant of the object is kept in memory past the destroy call. This can lead to an unexpected increase in memory consumption over the course of execution in applications that use temporary ESMF objects. For situations where the repeated creation and destruction of temporary objects leads to memory issues, it is recommended to call with noGarbage set to .TRUE., fully removing the entire temporary object from memory.

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

21.7.8 ESMF_StateGet - Get object-wide information from a State

INTERFACE:

```
! Private name; call using ESMF_StateGet()
subroutine ESMF_StateGetInfo(state, &
      itemSearch, itemorderflag, nestedFlag, &
      stateintent, itemCount, itemNameList, itemTypeList, name, rc)
```

ARGUMENTS:

```
   type(ESMF_State), intent(in) :: state
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
   character (len=*), intent(in), optional :: itemSearch
   type(ESMF_ItemOrder_Flag), intent(in), optional :: itemorderflag
   logical, intent(in), optional :: nestedFlag
   type(ESMF_StateIntent_Flag), intent(out), optional :: stateintent
   integer, intent(out), optional :: itemCount
   character (len=*), intent(out), optional :: itemNameList(:)
   type(ESMF_StateItem_Flag), intent(out), optional :: itemTypeList(:)
   character (len=*), intent(out), optional :: name
   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:

  6.1.0 Added argument `itemorderflag`. The new argument gives the user control over the order in which the items are returned.

DESCRIPTION:

Returns the requested information about this `ESMF_State`. The optional `itemSearch` argument may specify the name of an individual item to search for. When used in conjunction with the `nestedFlag`, nested States will also be searched.

Typically, an `ESMF_StateGet()` information request will be performed twice. The first time, the `itemCount` argument will be used to query the size of arrays that are needed. Arrays can then be allocated to the correct size for `itemNameList` and `itemtypeList` as needed. A second call to `ESMF_StateGet()` will then fill in the values.

The arguments are:

state An `ESMF_State` object to be queried.

[itemSearch] Query objects by name in the State. When the `nestedFlag` option is set to .true., all nested States will also be searched for the specified name.

[itemorderflag] Specifies the order of the returned items in the `itemNameList` and `itemTypeList`. The default is `ESMF_ITEMORDER_ABC`. See **??** for a full list of options.

[nestedFlag] When set to `.false.`, returns information at the current State level only (default) When set to `.true.`, additionally returns information from nested States

[stateintent] Returns the type, e.g., Import or Export, of this `ESMF_State`. Possible values are listed in Section 21.2.1.

[itemCount] Count of items in this `ESMF_State`. When the `nestedFlag` option is set to `.true.`, the count will include items present in nested States. When using `itemSearch`, it will count the number of items matching the specified name.

[itemNameList] Array of item names in this `ESMF_State`. When the `nestedFlag` option is set to `.true.`, the list will include items present in nested States. When using `itemSearch`, it will return the names of items matching the specified name. `itemNameList` must be at least `itemCount` long.

[itemTypeList] Array of possible item object types in this `ESMF_State`. When the `nestedFlag` option is set to `.true.`, the list will include items present in nested States. When using `itemSearch`, it will return the types of items matching the specified name. Must be at least `itemCount` long. Return values are listed in Section 21.2.2.

[name] Returns the name of this `ESMF_State`.

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

21.7.9 ESMF_StateGet - Get information about an item in a State by item name

INTERFACE:

```
! Private name; call using ESMF_StateGet()
subroutine ESMF_StateGetItemInfo(state, itemName, itemType, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len=*), intent(in) :: itemName
type(ESMF_StateItem_Flag), intent(out) :: itemType
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
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:

Returns the type for the item named `name` in this `ESMF_State`. If no item with this name exists, the value `ESMF_STATEITEM_NOTFOUND` will be returned and the error code will not be set to an error. Thus this routine can be used to safely query for the existence of items by name whether or not they are expected to be there. The error code will be set in case of other errors, for example if the `ESMF_State` itself is invalid.

The arguments are:

state ESMF_State to be queried.

itemName Name of the item to return information about.

itemType Returned item types for the item with the given name, including placeholder names. Options are listed in Section 21.2.2. If no item with the given name is found, ESMF_STATEITEM_NOTFOUND will be returned and rc will **not** be set to an error.

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

21.7.10 ESMF_StateGet - Get an item from a State by item name

INTERFACE:

```
subroutine ESMF_StateGet(state, itemName, <item>, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character (len=*), intent(in) :: itemName
<item>, see below for supported values
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
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:

Returns an <item> from an ESMF_State by item name. If the ESMF_State contains the <item> directly, only itemName is required.

If the state contains nested ESMF_States, the itemName argument may specify a fully qualified name to access the desired item with a single call. This is performed using the '/' character to separate the names of the intermediate State names leading to the desired item. (E.g., itemName='state1/state12/item').

Supported values for <item> are:

type(ESMF_Array), intent(out) :: array

type(ESMF_ArrayBundle), intent(out) :: arraybundle

type(ESMF_Field), intent(out) :: field

type(ESMF_FieldBundle), intent(out) :: fieldbundle

type(ESMF_RouteHandle), intent(out) :: routehandle

type(ESMF_State), intent(out) :: nestedState

The arguments are:

state State to query for an <item> named `itemName`.

itemName Name of <item> to be returned. This name may be fully qualified in order to access nested State items.

<item> Returned reference to the <item>.

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

21.7.11 ESMF_StateIsCreated - Check whether an State object has been created

INTERFACE:

```
   function ESMF_StateIsCreated(state, rc)
```

RETURN VALUE:

```
     logical :: ESMF_StateIsCreated
```

ARGUMENTS:

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

DESCRIPTION:

Return `.true.` if the `state` has been created. Otherwise return `.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:

state `ESMF_State` queried.

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

21.7.12 ESMF_StatePrint - Print State information

INTERFACE:

```
       subroutine ESMF_StatePrint(state, options, nestedFlag, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(in) :: state
character(len=*), intent(in), optional :: options
logical, intent(in), optional :: nestedFlag
integer, intent(out), optional :: rc
```

DESCRIPTION:

Prints information about the `state` to `stdout`.

The arguments are:

state The `ESMF_State` to print.

[options] Print options: " ", or "brief" - print names and types of the objects within the state (default), "long" - print additional information, such as proxy flags

[nestedFlag] When set to `.false.`, prints information about the current State level only (default), When set to `.true.`, additionally prints information from nested States

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

21.7.13 ESMF_StateRead – Read data items from a file into a State

INTERFACE:

```
subroutine ESMF_StateRead(state, fileName, rc)
```

ARGUMENTS:

```
type(ESMF_State), intent(inout) :: state
character (len=*), intent(in) :: fileName
integer, intent(out), optional :: rc
```

DESCRIPTION:

Currently limited to read in all Arrays from a NetCDF file and add them to a State object. Future releases will enable more items of a State to be read from a file of various formats.

Only PET 0 reads the file; the States in other PETs remain empty. Currently, the data is not decomposed or distributed; each PET has only 1 DE and only PET 0 contains data after reading the file. Future versions of ESMF will support data decomposition and distribution upon reading a file. See Section 21.3.7 for an example.

Note that the third party NetCDF library must be installed. For more details, see the "ESMF Users Guide", "Building and Installing the ESMF, Third Party Libraries, NetCDF" and the website http://www.unidata.ucar.edu/software/netcdf.

The arguments are:

state The `ESMF_State` to add items read from file. Currently only Arrays are supported.

fileName File to be read.

[rc] Return code; equals `ESMF_SUCCESS` if there are no errors. Equals `ESMF_RC_LIB_NOT_PRESENT` if the NetCDF library is not present.

21.7.14 ESMF_StateReconcile – Reconcile State data across all PETs in a VM

INTERFACE:

```
subroutine ESMF_StateReconcile(state, vm, rc)
```

ARGUMENTS:

```
type(ESMF_State),            intent(inout)          :: state
type(ESMF_VM),               intent(in),  optional :: vm
integer,                     intent(out), optional :: rc
```

DESCRIPTION:

Must be called for any `ESMF_State` which contains ESMF objects that have not been created on all the `PET`s of the currently running `ESMF_Component`. For example, if a coupler is operating on data which was created by another component that ran on only a subset of the couplers `PET`s, the coupler must make this call first before operating on any data inside that `ESMF_State`. After calling `ESMF_StateReconcile` all `PET`s will have a common view of all objects contained in this `ESMF_State`.

This call is collective across the specified VM.

The arguments are:

state `ESMF_State` to reconcile.

[vm] `ESMF_VM` for this `ESMF_Component`. By default, it is set to the current vm.

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

21.7.15 ESMF_StateRemove - Remove an item from a State - (DEPRECATED METHOD)

INTERFACE:

```
! Private name; call using ESMF_StateRemove ()
subroutine ESMF_StateRemoveOneItem (state, itemName, &
    relaxedFlag, rc)
```

ARGUMENTS:

```
  type(ESMF_State), intent(inout) :: state
  character(*), intent(in) :: itemName
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  logical, intent(in), optional :: relaxedFlag
  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.

- **DEPRECATED METHOD** as of ESMF 5.3.1. Please use ESMF_StateRemove, section 21.7.16 instead. Rationale: The list version is consistent with other ESMF container operations which use lists.

DESCRIPTION:

Remove an existing reference to an item from a State.

The arguments are:

state The ESMF_State within which itemName will be removed.

itemName The name of the item to be removed. This is a reference only. The item itself is unchanged.

If the state contains nested ESMF_States, the itemName argument may specify a fully qualified name to remove the desired item with a single call. This is performed using the "/" character to separate the names of the intermediate State names leading to the desired item. (E.g., itemName="state1/state12/item".

Since an item could potentially be referenced by multiple containers, it remains the responsibility of the user to manage its destruction when it is no longer in use.

[relaxedflag] A setting of .true. indicates a relaxed definition of "remove", where it is *not* an error if itemName is not present in the state. For .false. this is treated as an error condition. The default setting is .false..

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

21.7.16 ESMF_StateRemove - Remove a list of items from a State

INTERFACE:

```
! Private name; call using ESMF_StateRemove ()
subroutine ESMF_StateRemoveList (state, itemNameList, relaxedFlag, rc)
```

ARGUMENTS:

```
    type(ESMF_State), intent(inout) :: state
    character(*), intent(in) :: itemNameList(:)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    logical, intent(in), optional :: relaxedFlag
    integer, intent(out), optional :: rc
```

STATUS:

- This interface is backward compatible with ESMF versions starting at 5.3.1. If code using this interface compiles with any version of ESMF starting with 5.3.1, then it will compile with the current version.

DESCRIPTION:

Remove existing references to items from a State.

The arguments are:

state The ESMF_State within which itemName will be removed.

itemNameList The name of the items to be removed. This is a reference only. The items themselves are unchanged.

If the state contains nested ESMF_States, the itemName arguments may specify fully qualified names to remove the desired items with a single call. This is performed using the "/" character to separate the names of the intermediate State names leading to the desired items. (E.g., itemName="state1/state12/item".

Since items could potentially be referenced by multiple containers, it remains the responsibility of the user to manage their destruction when they are no longer in use.

[relaxedflag] A setting of .true. indicates a relaxed definition of "remove", where it is *not* an error if an item in the itemNameList is not present in the state. For .false. this is treated as an error condition. The default setting is .false..

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

21.7.17 ESMF_StateReplace - Replace a list of items within a State

INTERFACE:

```
   subroutine ESMF_StateReplace(state, <itemList>, relaxedflag, rc)
```

ARGUMENTS:

```
   type(ESMF_State), intent(inout) :: state
   <itemList>, see below for supported values
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
   logical, intent(in), optional :: relaxedflag
   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:

Replace a list of items with a ESMF_State. If an item in <itemList> does not match any items already present in state, an error is returned.

Supported values for <itemList> are:

type(ESMF_Array), intent(in) :: arrayList(:)

type(ESMF_ArrayBundle), intent(in) :: arraybundleList(:)

type(ESMF_Field), intent(in) :: fieldList(:)

type(ESMF_FieldBundle), intent(in) :: fieldbundleList(:)

type(ESMF_RouteHandle), intent(in) :: routehandleList(:)

type(ESMF_State), intent(in) :: nestedStateList(:)

The arguments are:

state An ESMF_State within which the <itemList> items will be replaced.

<itemList> The list of items to be replaced. This is a reference only; when the ESMF_State is destroyed the <itemList> contained in it will not be destroyed. Also, the items in the <itemList> cannot be safely destroyed before the ESMF_State is destroyed. Since <itemList> items can be added to multiple containers, it remains the responsibility of the user to manage their destruction when they are no longer in use.

[relaxedflag] A setting of .true. indicates a relaxed definition of "replace", where it is *not* an error if <itemList> contains items with names that are not found in state. The State is left unchanged for these items. For .false. this is treated as an error condition. The default setting is .false..

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

21.7.18 ESMF_StateSet - Set State aspects

INTERFACE:

```
   subroutine ESMF_StateSet(state, stateIntent, rc)
```

ARGUMENTS:

```
     type(ESMF_State),             intent(inout)         :: state
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_StateIntent_Flag), intent(in),  optional :: stateIntent
     integer,                     intent(out), optional :: rc
```

DESCRIPTION:

Set the info in the state object.

The arguments are:

state The ESMF_State to set.

stateIntent Intent, e.g. Import or Export, of this ESMF_State. Possible values are listed in Section 21.2.1.

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

21.7.19 ESMF_StateValidate - Check validity of a State

INTERFACE:

```
       subroutine ESMF_StateValidate(state, nestedFlag, rc)
```

ARGUMENTS:

```
      type(ESMF_State), intent(in) :: state
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      logical,          intent(in),  optional :: nestedFlag
      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:

Validates that the `state` is internally consistent. Currently this method determines if the `State` is uninitialized or already destroyed. The method returns an error code if problems are found.

The arguments are:

state The `ESMF_State` to validate.

[nestedFlag] `.false.` - validates at the current State level only (default) `.true.` - recursively validates any nested States

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

21.7.20 ESMF_StateWrite – Write items from a State to file

INTERFACE:

```
      subroutine ESMF_StateWrite(state, fileName, rc)
```

ARGUMENTS:

```
      type(ESMF_State),  intent(in)            :: state
      character (len=*), intent(in)            :: fileName
      integer,           intent(out), optional :: rc
```

DESCRIPTION:

Currently limited to write out all Arrays of a State object to a netCDF file. Future releases will enable more item types of a State to be written to files of various formats.

Writing is currently limited to PET 0; future versions of ESMF will allow parallel writing, as well as parallel reading.

See Section 21.3.7 for an example.

Note that the third party NetCDF library must be installed. For more details, see the "ESMF Users Guide", "Building and Installing the ESMF, Third Party Libraries, NetCDF" and the website http://www.unidata.ucar.edu/software/netcdf.

The arguments are:

state The ESMF_State from which to write items. Currently limited to Arrays.

fileName File to be written.

[rc] Return code; equals ESMF_SUCCESS if there are no errors. Equals ESMF_RC_LIB_NOT_PRESENT if the NetCDF library is not present.

22 Attachable Methods

22.1 Description

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

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

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

22.2 Use and Examples

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

22.2.1 Producer Component attaches user defined method

The producer Component attaches a user defined method to exportState during the Component's initialize method. The user defined method is attached with label finalCalculation by which it will become accessible to the consumer Component.

```
subroutine init(gcomp, importState, exportState, clock, rc)
  ! arguments
  type(ESMF_GridComp):: gcomp
  type(ESMF_State):: importState, exportState
  type(ESMF_Clock):: clock
  integer, intent(out):: rc

  rc = ESMF_SUCCESS
  call ESMF_MethodAdd(exportState, label="finalCalculation", &
    userRoutine=finalCalc, rc=rc)
  if (rc /= ESMF_SUCCESS) return

  ! just for testing purposes add the same method with a crazy string label
  call ESMF_MethodAdd(exportState, label="Somewhat of a SILLY @$^@_ label", &
    userRoutine=finalCalc, rc=rc)
  if (rc /= ESMF_SUCCESS) return
```

```
end subroutine !------------------------------------------------------------
```

22.2.2 Producer Component implements user defined method

The producer Component implements the attached, user defined method `finalCalc`. Strict interface rules apply for the user defined method.

```
subroutine finalCalc(state, rc)
  ! arguments
  type(ESMF_State):: state
  integer, intent(out):: rc

  rc = ESMF_SUCCESS

  ! access data objects in state and perform calculation
  print *, "dummy output from attached method "

end subroutine !------------------------------------------------------------
```

22.2.3 Consumer Component executes user defined method

The consumer Component executes the user defined method on the `importState`.

```
subroutine init(gcomp, importState, exportState, clock, rc)
  ! arguments
  type(ESMF_GridComp):: gcomp
  type(ESMF_State):: importState, exportState
  type(ESMF_Clock):: clock
  integer, intent(out):: rc

  integer:: userRc, i
  logical:: isPresent
  character(len=:), allocatable :: labelList(:)

  rc = ESMF_SUCCESS
```

The importState can be queried for a list of *all* the attached methods.

```
  call ESMF_MethodGet(importState, labelList=labelList, rc=rc)
  if (rc /= ESMF_SUCCESS) return

  ! print the labels
  do i=1, size(labelList)
    print *, labelList(i)
  enddo
```

It is also possible to check the importState whether a *specific* method is attached. This allows the consumer code to implement alternatives in case the method is not available.

```
    call ESMF_MethodGet(importState, label="finalCalculation", &
      isPresent=isPresent, rc=rc)
    if (rc /= ESMF_SUCCESS) return
```

Finally call into the attached method from the consumer side.

```
    call ESMF_MethodExecute(importState, label="finalCalculation", &
      userRc=userRc, rc=rc)
    if (rc /= ESMF_SUCCESS) return
    rc = userRc
    if (rc /= ESMF_SUCCESS) return

  end subroutine !------------------------------------------------------------
```

22.3 Restrictions and Future Work

1. **Not reconciled.** Attachable Methods are PET-local settings on an object. Currently Attachable Methods cannot be reconciled (i.e. ignored during `ESMF_StateReconcile()`).
2. **No copy nor move.** Currently Attachable Methods cannot be copied or moved between objects.

22.4 Class API

22.4.1 ESMF_MethodAdd - Attach user method to CplComp

INTERFACE:

```
   ! Private name; call using ESMF_MethodAdd()
   subroutine ESMF_MethodCplCompAdd(cplcomp, label, index, userRoutine, rc)
```

ARGUMENTS:

```
     type(ESMF_CplComp)                      :: cplcomp
     character(len=*), intent(in)            :: label
     integer,          intent(in),  optional :: index
     interface
       subroutine userRoutine(cplcomp, rc)
         use ESMF_CompMod
         implicit none
         type(ESMF_CplComp)          :: cplcomp      ! must not be optional
         integer, intent(out)        :: rc           ! must not be optional
       end subroutine
     end interface
     integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`. Error out if there is a previous attached method under the same `label` and `index`.

The arguments are:

cplcomp The `ESMF_CplComp` to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

userRoutine The user-supplied subroutine to be associated with the `label`.

The subroutine must have the exact interface shown above for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

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

22.4.2 ESMF_MethodAdd - Attach user method, located in shared object, to CplComp

INTERFACE:

```
! Private name; call using ESMF_MethodAdd()
subroutine ESMF_MethodCplCompAddShObj(cplcomp, label, index, userRoutine, &
  sharedObj, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                      :: cplcomp
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
character(len=*), intent(in)            :: userRoutine
character(len=*), intent(in),  optional :: sharedObj
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`. Error out if there is a previous attached method under the same `label` and `index`.

The arguments are:

cplcomp The `ESMF_CplComp` to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

userRoutine Name of user-supplied subroutine to be associated with the `label`, specified as a character string.

The subroutine must have the exact interface shown in `ESMF_MethodCplCompAdd` for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

[sharedObj] Name of shared object that contains `userRoutine`. If the `sharedObj` argument is not provided the executable itself will be searched for `userRoutine`.

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

22.4.3 ESMF_MethodAdd - Attach user method to GridComp

INTERFACE:

```
! Private name; call using ESMF_MethodAdd()
subroutine ESMF_MethodGridCompAdd(gcomp, label, index, userRoutine, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                     :: gcomp
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
interface
  subroutine userRoutine(gcomp, rc)
    use ESMF_CompMod
    implicit none
    type(ESMF_GridComp)         :: gcomp        ! must not be optional
    integer, intent(out)        :: rc           ! must not be optional
  end subroutine
end interface
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`. Error out if there is a previous attached method under the same `label` and `index`.

The arguments are:

gcomp The `ESMF_GridComp` to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

userRoutine The user-supplied subroutine to be associated with the `label`.

The subroutine must have the exact interface shown above for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

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

22.4.4 ESMF_MethodAdd - Attach user method, located in shared object, to GridComp

INTERFACE:

```
! Private name; call using ESMF_MethodAdd()
subroutine ESMF_MethodGridCompAddShObj(gcomp, label, index, userRoutine, &
  sharedObj, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                     :: gcomp
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
character(len=*), intent(in)            :: userRoutine
character(len=*), intent(in),  optional :: sharedObj
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`. Error out if there is a previous attached method under the same `label` and `index`.

The arguments are:

gcomp The `ESMF_GridComp` to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

userRoutine Name of user-supplied subroutine to be associated with the `label`, specified as a character string.

The subroutine must have the exact interface shown in `ESMF_MethodGridCompAdd` for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

[sharedObj] Name of shared object that contains `userRoutine`. If the `sharedObj` argument is not provided the executable itself will be searched for `userRoutine`.

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

22.4.5 ESMF_MethodAdd - Attach user method to State

INTERFACE:

```
! Private name; call using ESMF_MethodAdd()
subroutine ESMF_MethodStateAdd(state, label, index, userRoutine, rc)
```

ARGUMENTS:

```
type(ESMF_State)                          :: state
character(len=*), intent(in)              :: label
integer,          intent(in),  optional :: index
interface
  subroutine userRoutine(state, rc)
    use ESMF_StateMod
    implicit none
    type(ESMF_State)             :: state        ! must not be optional
    integer, intent(out)         :: rc           ! must not be optional
  end subroutine
end interface
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`. Error out if there is a previous attached method under the same `label` and `index`.

The arguments are:

state The `ESMF_State` to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

userRoutine The user-supplied subroutine to be associated with the `label`.

The subroutine must have the exact interface shown above for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

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

22.4.6 ESMF_MethodAdd - Attach user method, located in shared object, to State

INTERFACE:

```
! Private name; call using ESMF_MethodAdd()
subroutine ESMF_MethodStateAddShObj(state, label, index, userRoutine, &
  sharedObj, rc)
```

ARGUMENTS:

```
  type(ESMF_State)                        :: state
  character(len=*), intent(in)            :: label
  integer,          intent(in),  optional :: index
  character(len=*), intent(in)            :: userRoutine
  character(len=*), intent(in),  optional :: sharedObj
  integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`. Error out if there is a previous attached method under the same `label` and `index`.

The arguments are:

state The `ESMF_State` to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

userRoutine Name of user-supplied subroutine to be associated with the `label`, specified as a character string.

The subroutine must have the exact interface shown in `ESMF_MethodStateAdd` for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

[sharedObj] Name of shared object that contains `userRoutine`. If the `sharedObj` argument is not provided the executable itself will be searched for `userRoutine`.

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

22.4.7 ESMF_MethodAddReplace - Attach user method to CplComp

INTERFACE:

```
! Private name; call using ESMF_MethodAddReplace()
subroutine ESMF_MethodCplCompAddRep(cplcomp, label, index, userRoutine, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                      :: cplcomp
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
interface
  subroutine userRoutine(cplcomp, rc)
    use ESMF_CompMod
    implicit none
    type(ESMF_CplComp)          :: cplcomp      ! must not be optional
    integer, intent(out)        :: rc           ! must not be optional
  end subroutine
end interface
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`. Replacing potential previous attached method under the same `label` and `index`.

The arguments are:

cplcomp The `ESMF_CplComp` to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

userRoutine The user-supplied subroutine to be associated with the `label`.

The subroutine must have the exact interface shown above for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

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

22.4.8 ESMF_MethodAddReplace - Attach user method, located in shared object, to CplComp

INTERFACE:

```
! Private name; call using ESMF_MethodAddReplace()
subroutine ESMF_MethodCplCompAddRepShObj(cplcomp, label, index, userRoutine, &
  sharedObj, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                      :: cplcomp
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
character(len=*), intent(in)            :: userRoutine
character(len=*), intent(in),  optional :: sharedObj
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`. Replacing potential previous attached method under the same `label` and `index`.

The arguments are:

cplcomp The `ESMF_CplComp` to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

userRoutine Name of user-supplied subroutine to be associated with the `label`, specified as a character string.

The subroutine must have the exact interface shown in `ESMF_MethodCplCompAdd` for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

[sharedObj] Name of shared object that contains `userRoutine`. If the `sharedObj` argument is not provided the executable itself will be searched for `userRoutine`.

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

22.4.9 ESMF_MethodAddReplace - Attach user method to GridComp

INTERFACE:

```
! Private name; call using ESMF_MethodAddReplace()
subroutine ESMF_MethodGridCompAddRep(gcomp, label, index, userRoutine, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                     :: gcomp
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
interface
  subroutine userRoutine(gcomp, rc)
    use ESMF_CompMod
    implicit none
    type(ESMF_GridComp)         :: gcomp        ! must not be optional
    integer, intent(out)        :: rc           ! must not be optional
  end subroutine
end interface
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`. Replacing potential previous attached method under the same `label` and `index`.

The arguments are:

gcomp The ESMF_GridComp to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same label.

userRoutine The user-supplied subroutine to be associated with the label.

The subroutine must have the exact interface shown above for the userRoutine argument. Arguments in userRoutine must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

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

22.4.10 ESMF_MethodAddReplace - Attach user method, located in shared object, to GridComp

INTERFACE:

```
! Private name; call using ESMF_MethodAddReplace()
subroutine ESMF_MethodGridCompAddRepShObj(gcomp, label, index, userRoutine, &
  sharedObj, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                     :: gcomp
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
character(len=*), intent(in)            :: userRoutine
character(len=*), intent(in),  optional :: sharedObj
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach userRoutine. Replacing potential previous attached method under the same label and index.

The arguments are:

gcomp The ESMF_GridComp to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same label.

userRoutine Name of user-supplied subroutine to be associated with the label, specified as a character string.

The subroutine must have the exact interface shown in ESMF_MethodGridCompAdd for the userRoutine argument. Arguments in userRoutine must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another

procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

[sharedObj] Name of shared object that contains `userRoutine`. If the `sharedObj` argument is not provided the executable itself will be searched for `userRoutine`.

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

22.4.11 ESMF_MethodAddReplace - Attach user method to State

INTERFACE:

```
! Private name; call using ESMF_MethodAddReplace()
subroutine ESMF_MethodStateAddRep(state, label, index, userRoutine, rc)
```

ARGUMENTS:

```
type(ESMF_State)                          :: state
character(len=*), intent(in)              :: label
integer,          intent(in),  optional :: index
interface
  subroutine userRoutine(state, rc)
    use ESMF_StateMod
    implicit none
    type(ESMF_State)             :: state        ! must not be optional
    integer, intent(out)         :: rc           ! must not be optional
  end subroutine
end interface
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach `userRoutine`. Replacing potential previous attached method under the same `label` and `index`.

The arguments are:

state The `ESMF_State` to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

userRoutine The user-supplied subroutine to be associated with the `label`.

The subroutine must have the exact interface shown above for the `userRoutine` argument. Arguments in `userRoutine` must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

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

22.4.12 ESMF_MethodAddReplace - Attach user method, located in shared object, to State

INTERFACE:

```
! Private name; call using ESMF_MethodAddReplace()
subroutine ESMF_MethodStateAddRepShObj(state, label, index, userRoutine, &
  sharedObj, rc)
```

ARGUMENTS:

```
type(ESMF_State)                        :: state
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
character(len=*), intent(in)            :: userRoutine
character(len=*), intent(in),  optional :: sharedObj
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Attach userRoutine. Replacing potential previous attached method under the same label and index.

The arguments are:

state The ESMF_State to attach to.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same label.

userRoutine Name of user-supplied subroutine to be associated with the label, specified as a character string.

The subroutine must have the exact interface shown in ESMF_MethodStateAdd for the userRoutine argument. Arguments in userRoutine must not be declared as optional, and the types, intent and order must match. Prior to Fortran-2008, the subroutine must be either a module scope procedure, or an external procedure that has a matching interface block specified for it. An internal procedure which is contained within another procedure must not be used. From Fortran-2008 onwards, an internal procedure contained within either a main program or a module procedure may be used. If the internal procedure is contained within a module procedure, it is subject to initialization requirements. See: 16.4.9

[sharedObj] Name of shared object that contains userRoutine. If the sharedObj argument is not provided the executable itself will be searched for userRoutine.

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

22.4.13 ESMF_MethodExecute - Execute user method attached to CplComp

INTERFACE:

```
! Private name; call using ESMF_MethodExecute()
recursive subroutine ESMF_MethodCplCompExecute(cplcomp, label, index, existflag, &
  userRc, rc)
```

ARGUMENTS:

```
  type(ESMF_CplComp)                      :: cplcomp
  character(len=*), intent(in)            :: label
  integer,          intent(in),  optional :: index
  logical,          intent(out), optional :: existflag
  integer,          intent(out), optional :: userRc
  integer,          intent(out), optional :: rc
```

DESCRIPTION:

Execute attached method.

The arguments are:

cplcomp The `ESMF_CplComp` object holding the attachable method.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

[existflag] Returned `.true.` indicates that the method specified by `label` exists and was executed. A return value of `.false.` indicates that the method does not exist and consequently was not executed. By default, i.e. if `existflag` was not specified, the latter condition will lead to `rc` not equal `ESMF_SUCCESS` being returned. However, if `existflag` was specified, a method not existing is not an error condition.

[userRc] Return code set by attached method before returning.

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

22.4.14 ESMF_MethodExecute - Execute user method attached to GridComp

INTERFACE:

```
! Private name; call using ESMF_MethodExecute()
recursive subroutine ESMF_MethodGridCompExecute(gcomp, label, index, existflag, &
  userRc, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                     :: gcomp
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
logical,          intent(out), optional :: existflag
integer,          intent(out), optional :: userRc
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Execute attached method.

The arguments are:

gcomp The `ESMF_GridComp` object holding the attachable method.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

[existflag] Returned `.true.` indicates that the method specified by `label` exists and was executed. A return value of `.false.` indicates that the method does not exist and consequently was not executed. By default, i.e. if `existflag` was not specified, the latter condition will lead to `rc` not equal `ESMF_SUCCESS` being returned. However, if `existflag` was specified, a method not existing is not an error condition.

[userRc] Return code set by attached method before returning.

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

22.4.15 ESMF_MethodExecute - Execute user method attached to State

INTERFACE:

```
! Private name; call using ESMF_MethodExecute()
recursive subroutine ESMF_MethodStateExecute(state, label, index, existflag, &
  userRc, rc)
```

ARGUMENTS:

```
type(ESMF_State)                        :: state
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
logical,          intent(out), optional :: existflag
integer,          intent(out), optional :: userRc
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Execute attached method.

The arguments are:

state The ESMF_State object holding the attachable method.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same label.

[existflag] Returned .true. indicates that the method specified by label exists and was executed. A return value of .false. indicates that the method does not exist and consequently was not executed. By default, i.e. if existflag was not specified, the latter condition will lead to rc not equal ESMF_SUCCESS being returned. However, if existflag was specified, a method not existing is not an error condition.

[userRc] Return code set by attached method before returning.

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

22.4.16 ESMF_MethodGet - Get info about user method attached to CplComp

INTERFACE:

```
! Private name; call using ESMF_MethodGet()
subroutine ESMF_MethodCplCompGet(cplcomp, label, index, isPresent, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                      :: cplcomp
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
logical,          intent(out), optional :: isPresent
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Access information about attached method.

The arguments are:

cplcomp The ESMF_CplComp object holding the attachable method.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same label.

[isPresent] .true. if a method was attached for label/index. .false. otherwise.

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

22.4.17 ESMF_MethodGet - Get info about user methods attached to CplComp

INTERFACE:

```
! Private name; call using ESMF_MethodGet()
subroutine ESMF_MethodCplCompGetList(cplcomp, labelList, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                                  :: cplcomp
character(len=:), allocatable, intent(out)          :: labelList(:)
integer,                       intent(out), optional :: rc
```

DESCRIPTION:

Access labels of all attached methods.

The arguments are:

cplcomp The `ESMF_CplComp` object holding the attachable method.

labelList List of labels of *all* the attached methods. On return, it will be allocated with as many list elements as there are attached methods. The length of each label in `labelList` is that of the largest method label currently attached. Elements with shorter labels are padded with white spaces.

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

22.4.18 ESMF_MethodGet - Get info about user method attached to GridComp

INTERFACE:

```
! Private name; call using ESMF_MethodGet()
subroutine ESMF_MethodGridCompGet(gcomp, label, index, isPresent, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                    :: gcomp
character(len=*), intent(in)           :: label
integer,          intent(in),  optional :: index
logical,          intent(out), optional :: isPresent
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Access information about attached method.

The arguments are:

gcomp The ESMF_GridComp object holding the attachable method.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same label.

[isPresent] .true. if a method was attached for label/index. .false. otherwise.

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

22.4.19 ESMF_MethodGet - Get info about user methods attached to GridComp

INTERFACE:

```
! Private name; call using ESMF_MethodGet()
subroutine ESMF_MethodGridCompGetList(gcomp, labelList, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                                  :: gcomp
character(len=:), allocatable, intent(out)           :: labelList(:)
integer,                       intent(out), optional :: rc
```

DESCRIPTION:

Access labels of all attached methods.

The arguments are:

gcomp The ESMF_GridComp object holding the attachable method.

labelList List of labels of *all* the attached methods. On return, it will be allocated with as many list elements as there are attached methods. The length of each label in labelList is that of the largest method label currently attached. Elements with shorter labels are padded with white spaces.

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

22.4.20 ESMF_MethodGet - Get info about user method attached to State

INTERFACE:

```
! Private name; call using ESMF_MethodGet()
subroutine ESMF_MethodStateGet(state, label, index, isPresent, rc)
```

ARGUMENTS:

```
type(ESMF_State)                          :: state
character(len=*), intent(in)              :: label
integer,          intent(in),  optional :: index
logical,          intent(out), optional :: isPresent
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Access information about attached method.

The arguments are:

state The `ESMF_State` object holding the attachable method.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

[isPresent] `.true.` if a method was attached for `label`/`index`. `.false.` otherwise.

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

22.4.21 ESMF_MethodGet - Get info about user methods attached to State

INTERFACE:

```
! Private name; call using ESMF_MethodGet()
subroutine ESMF_MethodStateGetList(state, labelList, rc)
```

ARGUMENTS:

```
type(ESMF_State)                                    :: state
character(len=:), allocatable, intent(out)          :: labelList(:)
integer,                       intent(out), optional :: rc
```

DESCRIPTION:

Access labels of all attached methods.

The arguments are:

state The `ESMF_State` object holding the attachable method.

labelList List of labels of *all* the attached methods. On return, it will be allocated with as many list elements as there are attached methods. The length of each label in `labelList` is that of the largest method label currently attached. Elements with shorter labels are padded with white spaces.

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

22.4.22 ESMF_MethodRemove - Remove user method attached to CplComp

INTERFACE:

```
! Private name; call using ESMF_MethodRemove()
subroutine ESMF_MethodCplCompRemove(cplcomp, label, index, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                      :: cplcomp
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Remove attached method.

The arguments are:

cplcomp The `ESMF_CplComp` object holding the attachable method.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

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

22.4.23 ESMF_MethodRemove - Remove user method attached to GridComp

INTERFACE:

```
! Private name; call using ESMF_MethodRemove()
subroutine ESMF_MethodGridCompRemove(gcomp, label, index, rc)
```

ARGUMENTS:

```
type(ESMF_GridComp)                      :: gcomp
character(len=*),  intent(in)            :: label
 integer,          intent(in),  optional :: index
integer,           intent(out), optional :: rc
```

DESCRIPTION:

Remove attached method.

The arguments are:

gcomp The `ESMF_GridComp` object holding the attachable method.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

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

22.4.24 ESMF_MethodRemove - Remove user method attached to State

INTERFACE:

```
! Private name; call using ESMF_MethodRemove()
subroutine ESMF_MethodStateRemove(state, label, index, rc)
```

ARGUMENTS:

```
type(ESMF_State)                        :: state
character(len=*), intent(in)            :: label
integer,          intent(in),  optional :: index
integer,          intent(out), optional :: rc
```

DESCRIPTION:

Remove attached method.

The arguments are:

state The `ESMF_State` object holding the attachable method.

label Label of method.

[index] Integer modifier to distinguish multiple entries with the same `label`.

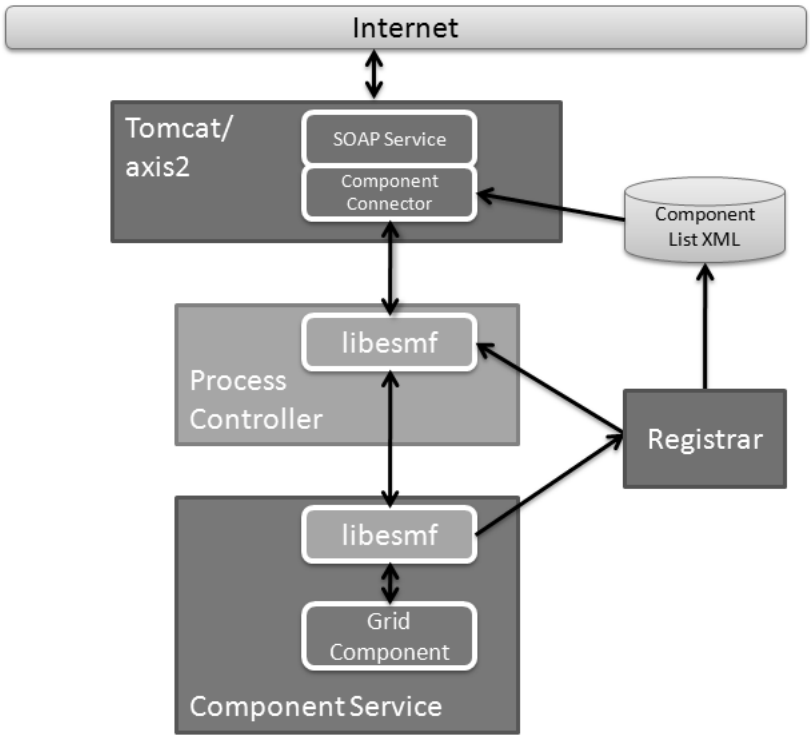
[rc] Return code; equals `ESMF_SUCCESS` if there are no errors.

23 Web Services

23.1 Description

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

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



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

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

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

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

23.1.1 Creating a Service around a Component

23.1.2 Code Modifications

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

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

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

```
use ESMF_WebServMod
....

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

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

23.1.3 Accessing the Service

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

clients to access the Component Service via the socket interface.

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

23.1.4 Client Application via C++ API

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

```
#include "ESMCI_WebServCompSvrClient.h"

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

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

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

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

23.1.5 Process Controller

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

23.1.6 Tomcat/Axis2

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

23.2 Use and Examples

The following examples demonstrate how to use ESMF Web Services.

23.2.1 Making a Component available through ESMF Web Services

In this example, a standard ESMF Component is made available through the Web Services interface.

The first step is to make sure your callback routines for initialize, run and finalize are setup. This is done by creating a register routine that sets the entry points for each of these callbacks. In this example, we've packaged it all up into a separate module.

```
module ESMF_WebServUserModel

  ! ESMF Framework module
  use ESMF

  implicit none

  public ESMF_WebServUserModelRegister

  contains

  !-------------------------------------------------------------------------
  !  The Registration routine
  !
  subroutine ESMF_WebServUserModelRegister(comp, rc)
    type(ESMF_GridComp)  :: comp
    integer, intent(out) :: rc

    ! Initialize return code
    rc = ESMF_SUCCESS

    print *, "User Comp1 Register starting"

    ! Register the callback routines.

    call ESMF_GridCompSetEntryPoint(comp, ESMF_METHOD_INITIALIZE, &
                                    userRoutine=user_init, rc=rc)
    if (rc/=ESMF_SUCCESS) return ! bail out

    call ESMF_GridCompSetEntryPoint(comp, ESMF_METHOD_RUN, &
                                    userRoutine=user_run, rc=rc)
    if (rc/=ESMF_SUCCESS) return ! bail out

    call ESMF_GridCompSetEntryPoint(comp, ESMF_METHOD_FINALIZE, &
                                    userRoutine=user_final, rc=rc)
    if (rc/=ESMF_SUCCESS) return ! bail out

    print *, "Registered Initialize, Run, and Finalize routines"
    print *, "User Comp1 Register returning"

  end subroutine
```

```
  !------------------------------------------------------------------------
  !  The Initialization routine
  !
  subroutine user_init(comp, importState, exportState, clock, rc)
    type(ESMF_GridComp)  :: comp
    type(ESMF_State)     :: importState, exportState
    type(ESMF_Clock)     :: clock
    integer, intent(out) :: rc

    ! Initialize return code
    rc = ESMF_SUCCESS

    print *, "User Comp1 Init"

  end subroutine user_init

  !------------------------------------------------------------------------
  !  The Run routine
  !
  subroutine user_run(comp, importState, exportState, clock, rc)
    type(ESMF_GridComp)  :: comp
    type(ESMF_State)     :: importState, exportState
    type(ESMF_Clock)     :: clock
    integer, intent(out) :: rc

    ! Initialize return code
    rc = ESMF_SUCCESS

    print *, "User Comp1 Run"

  end subroutine user_run

  !------------------------------------------------------------------------
  !  The Finalization routine
  !
  subroutine user_final(comp, importState, exportState, clock, rc)
    type(ESMF_GridComp)  :: comp
    type(ESMF_State)     :: importState, exportState
    type(ESMF_Clock)     :: clock
    integer, intent(out) :: rc

    ! Initialize return code
    rc = ESMF_SUCCESS

    print *, "User Comp1 Final"

  end subroutine user_final

end module ESMF_WebServUserModel
```

The actual driver code then becomes very simple; ESMF is initialized, the component is created, the callback functions for the component are registered, and the Web Service loop is started.

```
program WebServicesEx
```

```
#include "ESMF.h"

  ! ESMF Framework module
  use ESMF
  use ESMF_TestMod

  use ESMF_WebServMod
  use ESMF_WebServUserModel

  implicit none

  ! Local variables
  type(ESMF_GridComp) :: comp1     !! Grid Component
  integer             :: rc        !! Return Code
  integer             :: finalrc  !! Final return code
  integer             :: portNum   !! The port number for the listening socket
```

A listening socket will be created on the local machine with the specified port number. This socket is used by the service to wait for and receive requests from the client. Check with your system administrator to determine an appropriate port to use for your service.

```
  finalrc = ESMF_SUCCESS

  call ESMF_Initialize(defaultlogfilename="WebServicesEx.Log", &
                   logkindflag=ESMF_LOGKIND_MULTI, rc=rc)


  ! create the grid component
  comp1 = ESMF_GridCompCreate(name="My Component", rc=rc)


  ! Set up the register routine
  call ESMF_GridCompSetServices(comp1, &
          userRoutine=ESMF_WebServUserModelRegister, rc=rc)


  portNum = 27060

  ! Call the Web Services Loop and wait for requests to come in
  !call ESMF_WebServicesLoop(comp1, portNum, rc=rc)
```

The call to ESMF_WebServicesLoop will setup the listening socket for your service and will wait for requests from a client. As requests are received, the Web Services software will process the requests and then return to the loop to continue to wait.

The 3 main requests processed are INIT, RUN, and FINAL. These requests will then call the appropriate callback routine as specified in your register routine (as specified in the ESMF_GridCompSetServices call). In this example, when the INIT request is received, the user_init routine found in the ESMF_WebServUserModel module is called.

One other request is also processed by the Component Service, and that is the EXIT request. When this request is received, the Web Services loop is terminated and the remainder of the code after the ESMF_WebServicesLoop call is executed.

```
  call ESMF_Finalize(rc=rc)


end program WebServicesEx
```

23.3 Restrictions and Future Work

1. **Manual Control of Process.** Currently, the Component Service must be manually started and stopped. Future plans include having the Process Controller be responsible for controlling the Component Service processes.
2. **Data Streaming.** While data can be streamed from the web server to the client, it is not yet getting the data directly from the Component Service. Instead, the Component Service exports the data to a file which the Process Controller can read and return across the network interface. The data streaming capabilities will be a major component of future improvements to the Web Services architecture.

23.4 Class API

23.4.1 ESMF_WebServicesLoop

INTERFACE:

```
   subroutine ESMF_WebServicesLoop(comp, portNum, clientId, registrarHost, rc)
```

ARGUMENTS:

```
     type(ESMF_GridComp)                          :: comp
     integer,            intent(inout), optional :: portNum
     character(len=*),   intent(in),    optional, target :: clientId
     character(len=*),   intent(in),    optional, target :: registrarHost
     integer,            intent(out),   optional :: rc
```

DESCRIPTION:

Encapsulates all of the functionality necessary to setup a component as a component service. On the root PET, it registers the component service and then enters into a loop that waits for requests on a socket. The loop continues until an "exit" request is received, at which point it exits the loop and unregisters the service. On any PET other than the root PET, it sets up a process block that waits for instructions from the root PET. Instructions will come as requests are received from the socket.

The arguments are:

[comp] `ESMF_CplComp` object that represents the Grid Component for which routine is run.

[portNum] Number of the port on which the component service is listening.

[clientId] Identifier of the client responsible for this component service. If a Process Controller application manages this component service, then the clientId is provided to the component service application in the command line. Otherwise, the clientId is not necessary.

[registrarHost] Name of the host on which the Registrar is running. Needed so the component service can notify the Registrar when it is ready to receive requests from clients.

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

23.4.2 ESMF_WebServicesCplCompLoop

INTERFACE:

```
subroutine ESMF_WebServicesCplCompLoop(comp, portNum, clientId, registrarHost, rc)
```

ARGUMENTS:

```
type(ESMF_CplComp)                          :: comp
integer,           intent(inout), optional :: portNum
character(len=*),  intent(in),    optional, target :: clientId
character(len=*),  intent(in),    optional, target :: registrarHost
integer,           intent(out),   optional :: rc
```

DESCRIPTION:

Encapsulates all of the functionality necessary to setup a component as a component service. On the root PET, it registers the component service and then enters into a loop that waits for requests on a socket. The loop continues until an "exit" request is received, at which point it exits the loop and unregisters the service. On any PET other than the root PET, it sets up a process block that waits for instructions from the root PET. Instructions will come as requests are received from the socket.

The arguments are:

[comp] `ESMF_CplComp` object that represents the Grid Component for which routine is run.

[portNum] Number of the port on which the component service is listening.

[clientId] Identifier of the client responsible for this component service. If a Process Controller application manages this component service, then the clientId is provided to the component service application in the command line. Otherwise, the clientId is not necessary.

[registrarHost] Name of the host on which the Registrar is running. Needed so the component service can notify the Registrar when it is ready to receive requests from clients.

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

Part IV

Infrastructure: Fields and Grids

24 Overview of Data Classes

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

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

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

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

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

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

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

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

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

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

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

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

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

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

24.1 Bit-for-Bit Considerations

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

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

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

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

24.2 Regrid

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

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

There are two options for accessing ESMF regridding functionality: **offline** and **integrated**. Offline regridding is a process whereby interpolation weights are generated by a separate ESMF command line tool, not within the user code. The ESMF offline regridding tool also only generates the interpolation matrix, the user is responsible for reading in this matrix and doing the actual interpolation (multiplication by the sparse matrix) in their code. Please see Section 12 for a description of the offline regridding command line tool and the options it supports. For user convenience, there is also a method interface to the offline regrid tool functionality which is described in Section 24.3.1. In contrast to offline regridding, integrated regridding is a process whereby interpolation weights are generated via subroutine calls during the execution of the user's code. In addition to generating the weights, integrated regridding can also produce a **RouteHandle** (described in Section **??**) which allows the user to perform the parallel sparse matrix multiplication using ESMF methods. In other words, ESMF integrated regridding allows a user to perform the whole process of interpolation within their code.

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

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

Name	Access via	Inputs	Outputs		Description
			Weights	RouteHandle	
ESMF_FieldRegridStore()	Subroutine call	Field object	yes	yes	Sec. 26.6.60
ESMF_FieldBundleRegridStore()	Subroutine call	Fieldbundle obj.	no	yes	Sec. 25.5.26
ESMF_RegridWeightGen()	Subroutine call	Grid files	yes	no	Sec. 24.3.1
ESMF_RegridWeightGen	Command Line Tool	Grid files	yes	no	Sec. 12

Table 1: Regrid Interfaces

24.2.1 Interpolation methods: bilinear

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

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

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

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

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

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

Restrictions:

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

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

24.2.2 Interpolation methods: higher-order patch

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

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

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

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

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

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

- NONE

Restrictions:

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

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

24.2.3 Interpolation methods: nearest source to destination

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

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

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

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

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

Restrictions:
NONE

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

24.2.4 Interpolation methods: nearest destination to source

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

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

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

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

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

Restrictions:

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

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

24.2.5 Interpolation methods: first-order conservative

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

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

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

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

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

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

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

Restrictions:

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

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

24.2.6 Interpolation methods: second-order conservative

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

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

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

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

Where:

s_i is the intersecting source cell value.

∇s_i is the intersecting source cell gradient.

c_{si} is the intersecting source cell centroid.

c_d is the destination cell centroid.

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

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

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

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

- NONE

Restrictions:

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

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

24.2.7 Conservation

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

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

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

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

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

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

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

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

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

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

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

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

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

24.2.9 Great circle cells

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

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

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

24.2.10 Masking

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

The user may mask out points in the source Field or destination Field or both. To do masking the user sets mask information in the Grid (see 31.3.17), Mesh (see **??**), or LocStream (see **??**) upon which the Fields passed into the `ESMF_FieldRegridStore()` call are built. The `srcMaskValues` and `dstMaskValues` arguments to that call can then be used to specify which values in that mask information indicate that a location should be masked out. For example, if `dstMaskValues` is set to (/1,2/), then any location that has a value of 1 or 2 in the mask information of the Grid, Mesh or LocStream upon which the destination Field is built will be masked out.

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

24.2.11 Extrapolation methods: overview

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

24.2.12 Extrapolation methods: nearest source to destination

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

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

24.2.13 Extrapolation methods: inverse distance weighted average

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

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

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

24.2.14 Extrapolation methods: creep fill

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

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

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

24.2.15 Unmapped destination points

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

24.2.16 Spherical grids and poles

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

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

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

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

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

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

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

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

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

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

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

24.2.17 Troubleshooting guide

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

Problem: Regridding is too slow.

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

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

In particular:

- If Grid coordinates do not change during a run, then the `ESMF_FieldRegridStore()` call can be done once between a pair of Fields at the beginning and the resulting routeHandle used for each timestep during the run.
- If a pair of Fields was created with exactly the same arguments to `ESMF_FieldCreate()` as the pair of Fields used during an `ESMF_FieldRegridStore()` call, then the resulting routeHandle can also be used between that pair of Fields.

Problem: Distortions in destination Field at periodic boundary.

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

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

24.2.18 Restrictions and Future Work

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

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

24.2.19 Design and implementation notes

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

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

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

- The creation of a mesh composed of triangles and quadrilaterals or hexahedrons and tetrahedrons.
- The object relations data base to store the connections between objects (e.g. which element contains which nodes).
- The fields to hold data (e.g. coordinates). We currently only build fields on nodes and elements (2D and 3D).
- Iteration to move through mesh entities.
- The parallel code to maintain information about the distribution of the mesh across processors and to communicate data between parts of the mesh on different processors (i.e. halos).

24.3 File-based Regrid API

24.3.1 ESMF_RegridWeightGen - Generate regrid weight file from grid files

INTERFACE:

```
! Private name; call using ESMF_RegridWeightGen()
subroutine ESMF_RegridWeightGenFile(srcFile, dstFile, &
  weightFile, rhFile, regridmethod, polemethod, regridPoleNPnts, lineType, normType, &
  extrapMethod, extrapNumSrcPnts, extrapDistExponent, extrapNumLevels, &
  unmappedaction, ignoreDegenerate, srcFileType, dstFileType, &
  srcRegionalFlag, dstRegionalFlag, srcMeshname, dstMeshname,  &
```

```
    srcMissingvalueFlag, srcMissingvalueVar, &
    dstMissingvalueFlag, dstMissingvalueVar, &
    useSrcCoordFlag, srcCoordinateVars, &
    useDstCoordFlag, dstCoordinateVars, &
    useSrcCornerFlag, useDstCornerFlag, &
    useUserAreaFlag, largefileFlag, &
    netcdf4fileFlag, weightOnlyFlag, &
    tileFilePath, &
    verboseFlag, rc)
```

ARGUMENTS:

```
  character(len=*),             intent(in)            :: srcFile
  character(len=*),             intent(in)            :: dstFile
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
  character(len=*),             intent(in),  optional :: weightFile
  character(len=*),             intent(in),  optional :: rhFile
  type(ESMF_RegridMethod_Flag), intent(in),  optional :: regridmethod
  type(ESMF_PoleMethod_Flag),   intent(in),  optional :: polemethod
  integer,                      intent(in),  optional :: regridPoleNPnts
  type(ESMF_LineType_Flag),     intent(in),  optional :: lineType
  type(ESMF_NormType_Flag),     intent(in),  optional :: normType
  type(ESMF_ExtrapMethod_Flag),   intent(in),    optional :: extrapMethod
  integer,                        intent(in),    optional :: extrapNumSrcPnts
  real,                           intent(in),    optional :: extrapDistExponent
  integer,                      intent(in), optional :: extrapNumLevels
  type(ESMF_UnmappedAction_Flag),intent(in), optional :: unmappedaction
  logical,                      intent(in),  optional :: ignoreDegenerate
  type(ESMF_FileFormat_Flag),   intent(in),  optional :: srcFileType
  type(ESMF_FileFormat_Flag),   intent(in),  optional :: dstFileType
  logical,                      intent(in),  optional :: srcRegionalFlag
  logical,                      intent(in),  optional :: dstRegionalFlag
  character(len=*),             intent(in),  optional :: srcMeshname
  character(len=*),             intent(in),  optional :: dstMeshname
  logical,                      intent(in),  optional :: srcMissingValueFlag
  character(len=*),             intent(in),  optional :: srcMissingValueVar
  logical,                      intent(in),  optional :: dstMissingValueFlag
  character(len=*),             intent(in),  optional :: dstMissingValueVar
  logical,                      intent(in),  optional :: useSrcCoordFlag
  character(len=*),             intent(in),  optional :: srcCoordinateVars(:)
  logical,                      intent(in),  optional :: useDstCoordFlag
  character(len=*),             intent(in),  optional :: dstCoordinateVars(:)
  logical,                      intent(in),  optional :: useSrcCornerFlag
  logical,                      intent(in),  optional :: useDstCornerFlag
  logical,                      intent(in),  optional :: useUserAreaFlag
  logical,                      intent(in),  optional :: largefileFlag
  logical,                      intent(in),  optional :: netcdf4fileFlag
  logical,                      intent(in),  optional :: weightOnlyFlag
  logical,                      intent(in),  optional :: verboseFlag
  character(len=*),             intent(in),  optional :: tileFilePath
  integer,                      intent(out), optional :: rc
```

DESCRIPTION:

This subroutine provides the same function as the ESMF_RegridWeightGen application described in Section 12. It takes two grid files in NetCDF format and writes out an interpolation weight file also in NetCDF format. The interpolation weights can be generated with the bilinear (24.2.1), higher-order patch (24.2.2), or first order conservative (24.2.5) methods. The grid files can be in one of the following four formats:

- The SCRIP format (12.8.1)
- The native ESMF format for an unstructured grid (12.8.2)
- The CF Convention Single Tile File format (12.8.3)
- The proposed CF Unstructured grid (UGRID) format (12.8.4)
- The GRIDSPEC Mosaic File format (12.8.5)

The weight file is created in SCRIP format (12.9). The optional arguments allow users to specify various options to control the regrid operation, such as which pole option to use, whether to use user-specified area in the conservative regridding, or whether ESMF should generate masks using a given variable's missing value. There are also optional arguments specific to a certain type of the grid file. All the optional arguments are similar to the command line arguments for the ESMF_RegridWeightGen application (12.6). The acceptable values and the default value for the optional arguments are listed below.

The arguments are:

srcFile The source grid file name.

dstFile The destination grid file name.

weightFile The interpolation weight file name.

[rhFile] The RouteHandle file name.

[regridmethod] The type of interpolation. Please see Section **??** for a list of valid options. If not specified, defaults to ESMF_REGRIDMETHOD_BILINEAR.

[polemethod] A flag to indicate which type of artificial pole to construct on the source Grid for regridding. Please see Section **??** for a list of valid options. The default value varies depending on the regridding method and the grid type and format.

[regridPoleNPnts] If polemethod is set to ESMF_POLEMETHOD_NPNTAVG, this argument is required to specify how many points should be averaged over at the pole.

[lineType] This argument controls 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. As would be expected, this argument is only applicable when srcField and dstField are built on grids which lie on the surface of a sphere. Section **??** shows a list of valid options for this argument. If not specified, the default depends on the regrid method. Section **??** has the defaults by line type. Figure 24.2.16 shows which line types are supported for each regrid method as well as showing the default line type by regrid method.

[normType] This argument controls the type of normalization used when generating conservative weights. This option only applies to weights generated with regridmethod=ESMF_REGRIDMETHOD_CONSERVE. Please see Section **??** for a list of valid options. If not specified normType defaults to ESMF_NORMTYPE_DSTAREA.

[extrapMethod] The type of extrapolation. Please see Section **??** for a list of valid options. If not specified, defaults to ESMF_EXTRAPMETHOD_NONE.

[extrapNumSrcPnts] The number of source points to use for the extrapolation methods that use more than one source point (e.g. ESMF_EXTRAPMETHOD_NEAREST_IDAVG). If not specified, defaults to 8.

[extrapDistExponent] The exponent to raise the distance to when calculating weights for the `ESMF_EXTRAPMETHOD_NEAREST_IDAVG` extrapolation method. A higher value reduces the influence of more distant points. If not specified, defaults to 2.0.

[unmappedaction] Specifies what should happen if there are destination points that can't be mapped to a source cell. Please see Section **??** for a list of valid options. If not specified, `unmappedaction` defaults to `ESMF_UNMAPPEDACTION_ERROR`.

[ignoreDegenerate] Ignore degenerate cells when checking the input Grids or Meshes for errors. If this is set to true, then the regridding proceeds, but degenerate cells will be skipped. If set to false, a degenerate cell produces an error. If not specified, `ignoreDegenerate` defaults to false.

[srcFileType] The file format of the source grid. Please see Section **??** for a list of valid options. If not specifed, the program will determine the file format automatically.

[dstFileType] The file format of the destination grid. Please see Section **??** for a list of valid options. If not specifed, the program will determine the file format automatically.

[srcRegionalFlag] If .TRUE., the source grid is a regional grid, otherwise, it is a global grid. The default value is .FALSE.

[dstRegionalFlag] If .TRUE., the destination grid is a regional grid, otherwise, it is a global grid. The default value is .FALSE.

[srcMeshname] If the source file is in UGRID format, this argument is required to define the dummy variable name in the grid file that contains the mesh topology info.

[dstMeshname] If the destination file is in UGRID format, this argument is required to define the dummy variable name in the grid file that contains the mesh topology info.

[srcMissingValueFlag] If .TRUE., the source grid mask will be constructed using the missing values of the variable defined in `srcMissingValueVar`. This flag is only used for the grid defined in the GRIDSPEC or the UGRID file formats. The default value is .FALSE..

[srcMissingValueVar] If `srcMissingValueFlag` is .TRUE., the argument is required to define the variable name whose missing values will be used to construct the grid mask. It is only used for the grid defined in the GRIDSPEC or the UGRID file formats.

[dstMissingValueFlag] If .TRUE., the destination grid mask will be constructed using the missing values of the variable defined in `dstMissingValueVar`. This flag is only used for the grid defined in the GRIDSPEC or the UGRID file formats. The default value is .FALSE..

[dstMissingValueVar] If `dstMissingValueFlag` is .TRUE., the argument is required to define the variable name whose missing values will be used to construct the grid mask. It is only used for the grid defined in the GRIDSPEC or the UGRID file formats.

[useSrcCoordFlag] If .TRUE., the coordinate variables defined in `srcCoordinateVars` will be used as the longitude and latitude variables for the source grid. This flag is only used for the GRIDSPEC file format. The default is .FALSE.

[srcCoordinateVars] If `useSrcCoordFlag` is .TRUE., this argument defines the longitude and ! latitude variables in the source grid file to be used for the regrid. This argument is only used when the grid file is in GRIDSPEC format. `srcCoordinateVars` should be a array of 2 elements.

[useDstCoordFlag] If .TRUE., the coordinate variables defined in `dstCoordinateVars` will be used as the longitude and latitude variables for the destination grid. This flag is only used for the GRIDSPEC file format. The default is .FALSE.

[dstCoordinateVars] If `useDstCoordFlag` is .TRUE., this argument defines the longitude and latitude variables in the destination grid file to be used for the regrid. This argument is only used when the grid file is in GRIDSPEC format. `dstCoordinateVars` should be a array of 2 elements.

[useSrcCornerFlag] If `useSrcCornerFlag` is .TRUE., the corner coordinates of the source file will be used for regridding. Otherwise, the center coordinates will be us ed. The default is .FALSE. The corner stagger is not supported for the SCRIP formatted input grid or multi-tile GRIDSPEC MOSAIC input grid.

[useDstCornerFlag] If `useDstCornerFlag` is .TRUE., the corner coordinates of the destination file will be used for regridding. Otherwise, the center coordinates will be used. The default is .FALSE. The corner stagger is not supported for the SCRIP formatted input grid or multi-tile GRIDSPEC MOSAIC input grid.

[useUserAreaFlag] If .TRUE., the element area values defined in the grid files are used. Only the SCRIP and ESMF format grid files have user specified areas. This flag is only used for conservative regridding. The default is .FALSE..

[largefileFlag] If .TRUE., the output weight file is in NetCDF 64bit offset format. The default is .FALSE..

[netcdf4fileFlag] If .TRUE., the output weight file is in NetCDF4 file format. The default is .FALSE..

[weightOnlyFlag] If .TRUE., the output weight file only contains factorList and factorIndexList. The default is .FALSE..

[verboseFlag] If .TRUE., it will print summary information about the regrid parameters, default to .FALSE..

[tileFilePath] Optional argument to define the path where the tile files reside. If it is given, it overwrites the path defined in `gridlocation` variable in the mosaic file.

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

24.3.2 ESMF_RegridWeightGen - Generate regrid routeHandle and an optional weight file from grid files with user-specified distribution

INTERFACE:

```
! Private name; call using ESMF_RegridWeightGen()
subroutine ESMF_RegridWeightGenDG(srcFile, dstFile, regridRouteHandle, &
  srcElementDistgrid, dstElementDistgrid, &
  srcNodalDistgrid, dstNodalDistgrid, &
  weightFile, regridmethod, lineType, normType, &
  extrapMethod, extrapNumSrcPnts, extrapDistExponent, extrapNumLevels,&
  unmappedaction, ignoreDegenerate, useUserAreaFlag, &
  largefileFlag, netcdf4fileFlag, &
  weightOnlyFlag, verboseFlag, rc)
```

ARGUMENTS:

```
character(len=*),             intent(in)            :: srcFile
character(len=*),             intent(in)            :: dstFile
type(ESMF_RouteHandle),       intent(out)           :: regridRouteHandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_DistGrid),          intent(in),  optional :: srcElementDistgrid
type(ESMF_DistGrid),          intent(in),  optional :: dstElementDistgrid
character(len=*),             intent(in),  optional :: weightFile
type(ESMF_DistGrid),          intent(in),  optional :: srcNodalDistgrid
```

```
type(ESMF_DistGrid),           intent(in),  optional :: dstNodalDistgrid
type(ESMF_RegridMethod_Flag), intent(in),  optional :: regridmethod
type(ESMF_LineType_Flag),      intent(in),  optional :: lineType
type(ESMF_NormType_Flag),      intent(in),  optional :: normType
type(ESMF_ExtrapMethod_Flag),   intent(in),    optional :: extrapMethod
integer,                        intent(in),    optional :: extrapNumSrcPnts
real,                           intent(in),    optional :: extrapDistExponent
integer,                       intent(in),  optional :: extrapNumLevels
type(ESMF_UnmappedAction_Flag),intent(in), optional :: unmappedaction
logical,                       intent(in),  optional :: ignoreDegenerate
logical,                       intent(in),  optional :: useUserAreaFlag
logical,                       intent(in),  optional :: largefileFlag
logical,                       intent(in),  optional :: netcdf4fileFlag
logical,                       intent(in),  optional :: weightOnlyFlag
logical,                       intent(in),  optional :: verboseFlag
integer,                       intent(out), optional :: rc
```

DESCRIPTION:

This subroutine does online regridding weight generation from files with user specified distribution. The main differences between this API and the one in 24.3.1 are listed below:

- The input grids are always represented as `ESMF_Mesh` whether they are logically rectangular or unstructured.
- The input grids will be decomposed using a user-specified distribution instead of a fixed decomposition in the other subroutine if `srcElementDistgrid` and `dstElementDistgrid` are specified.
- The source and destination grid files have to be in the SCRIP grid file format.
- This subroutine has one additional required argument `regridRouteHandle` and four additional optional arguments: `srcElementDistgrid`, `dstElementDistgrid`, `srcNodelDistgrid` and `dstNodalDistgrid`. These four arguments are of type `ESMF_DistGrid`, they are used to define the distribution of the source and destination grid elements and nodes. The output `regridRouteHandle` allows users to regrid the field values later in the application.
- The `weightFile` argument is optional. When it is given, a weightfile will be generated as well.

The arguments are:

srcFile The source grid file name in SCRIP grid file format

dstFile The destination grid file name in SCRIP grid file format

regridRouteHandle The regrid RouteHandle returned by `ESMF_FieldRegridStore()`

srcElementDistgrid An optional distGrid that specifies the distribution of the source grid's elements. If not specified, a system-defined block decomposition is used.

dstElementDistgrid An optional distGrid that specifies the distribution of the destination grid's elements. If not specified, a system-defined block decomposition is used.

weightFile The interpolation weight file name. If present, an output weight file will be generated.

srcNodalDistgrid An optional distGrid that specifies the distribution of the source grid's nodes

dstNodalDistgrid An optional distGrid that specifies the distribution of the destination grid's nodes

[regridmethod] The type of interpolation. Please see Section **??** for a list of valid options. If not specified, defaults to ESMF_REGRIDMETHOD_BILINEAR.

[lineType] This argument controls 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. As would be expected, this argument is only applicable when srcField and dstField are built on grids which lie on the surface of a sphere. Section **??** shows a list of valid options for this argument. If not specified, the default depends on the regrid method. Section **??** has the defaults by line type. Figure 24.2.16 shows which line types are supported for each regrid method as well as showing the default line type by regrid method.

[normType] This argument controls the type of normalization used when generating conservative weights. This option only applies to weights generated with regridmethod=ESMF_REGRIDMETHOD_CONSERVE. Please see Section **??** for a list of valid options. If not specified normType defaults to ESMF_NORMTYPE_DSTAREA.

[extrapMethod] The type of extrapolation. Please see Section **??** for a list of valid options. If not specified, defaults to ESMF_EXTRAPMETHOD_NONE.

[extrapNumSrcPnts] The number of source points to use for the extrapolation methods that use more than one source point (e.g. ESMF_EXTRAPMETHOD_NEAREST_IDAVG). If not specified, defaults to 8..

[extrapDistExponent] The exponent to raise the distance to when calculating weights for the ESMF_EXTRAPMETHOD_NEAREST_IDAVG extrapolation method. A higher value reduces the influence of more distant points. If not specified, defaults to 2.0.

[unmappedaction] Specifies what should happen if there are destination points that can't be mapped to a source cell. Please see Section **??** for a list of valid options. If not specified, unmappedaction defaults to ESMF_UNMAPPEDACTION_ERROR.

[ignoreDegenerate] Ignore degenerate cells when checking the input Grids or Meshes for errors. If this is set to true, then the regridding proceeds, but degenerate cells will be skipped. If set to false, a degenerate cell produces an error. If not specified, ignoreDegenerate defaults to false.

[useUserAreaFlag] If .TRUE., the element area values defined in the grid files are used. Only the SCRIP and ESMF format grid files have user specified areas. This flag is only used for conservative regridding. The default is .FALSE.

[largefileFlag] If .TRUE., the output weight file is in NetCDF 64bit offset format. The default is .FALSE.

[netcdf4fileFlag] If .TRUE., the output weight file is in NetCDF4 file format. The default is .FALSE.

[weightOnlyFlag] If .TRUE., the output weight file only contains factorList and factorIndexList. The default is .FALSE.

[verboseFlag] If .TRUE., it will print summary information about the regrid parameters, default to .FALSE.

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

24.3.3 ESMF_FileRegrid - Regrid variables defined in the grid files

INTERFACE:

```
subroutine ESMF_FileRegrid(srcFile, dstFile, srcVarName, dstVarName, &
  dstLoc, srcDataFile, dstDataFile, tileFilePath, &
  dstCoordVars, regridmethod, polemethod, regridPoleNPnts, &
  unmappedaction, ignoreDegenerate, srcRegionalFlag, dstRegionalFlag, &
  verboseFlag, rc)
```

ARGUMENTS:

```
  character(len=*),             intent(in)            :: srcFile
  character(len=*),             intent(in)            :: dstFile
  character(len=*),             intent(in)            :: srcVarName
  character(len=*),             intent(in)            :: dstVarName
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  character(len=*),             intent(in),  optional :: dstLoc
  character(len=*),             intent(in),  optional :: srcDataFile
  character(len=*),             intent(in),  optional :: dstDataFile
  character(len=*),             intent(in),  optional :: tileFilePath
  character(len=*),             intent(in),  optional :: dstCoordVars
  type(ESMF_RegridMethod_Flag), intent(in),  optional :: regridmethod
  type(ESMF_PoleMethod_Flag),   intent(in),  optional :: polemethod
  integer,                      intent(in),  optional :: regridPoleNPnts
  type(ESMF_UnmappedAction_Flag),intent(in), optional :: unmappedaction
  logical,                      intent(in),  optional :: ignoreDegenerate
  logical,                      intent(in),  optional :: srcRegionalFlag
  logical,                      intent(in),  optional :: dstRegionalFlag
  logical,                      intent(in),  optional :: verboseFlag
  integer,                      intent(out), optional :: rc
```

DESCRIPTION:

This subroutine provides the same function as the `ESMF_Regrid` application described in Section 13. It takes two grid files in NetCDF format and interpolate the variable defined in the source grid file to the destination variable using one of the ESMF supported regrid methods – bilinear (24.2.1), higher-order patch (24.2.2), first order conservative (24.2.5) or nearest neighbor methods. The grid files can be in one of the following two formats:

- The GRIDSPEC Tile grid file following the CF metadata convention (12.8.3) for logically rectangular grids
- The proposed CF Unstructured grid (UGRID) format (12.8.4) for unstructured grids.

The optional arguments allow users to specify various options to control the regrid operation, such as which pole option to use, or whether to use user-specified area in the conservative regridding. The acceptable values and the default value for the optional arguments are listed below.

The arguments are:

srcFile The source grid file name.

dstFile The destination grid file name.

srcVarName The source variable names to be regridded. If more than one, separate them by comma.

dstVarName The destination variable names to be regridded to. If more than one, separate them by comma.

[dstLoc] The destination variable's location, either 'node' or 'face'. This argument is only used when the destination grid file is UGRID, the regridding method is non-conservative and the destination variable does not exist in the destination grid file. If not specified, default is 'face'.

[srcDataFile] The input data file prefix if the srcFile is in GRIDSPEC MOSAIC fileformat. The tilename and the file extension (.nc) will be added to the prefix. The tilename is defined in the MOSAIC file using variable "gridtiles".

[dstDataFile] The output data file prefix if the dstFile is in GRIDSPEC MOSAIC fileformat. The tilename and the file extension (.nc) will be added to the prefix. The tilename is defined in the MOSAIC file using variable "gridtiles".

[tileFilePath] The alternative file path for the tile files and mosaic data files when either srcFile or dstFile 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.

[dstCoordVars] The destination coordinate variable names if the dstVarName does not exist in the dstFile

[regridmethod] The type of interpolation. Please see Section **??** for a list of valid options. If not specified, defaults to `ESMF_REGRIDMETHOD_BILINEAR`.

[polemethod] A flag to indicate which type of artificial pole to construct on the source Grid for regridding. Please see Section **??** for a list of valid options. The default value varies depending on the regridding method and the grid type and format.

[regridPoleNPnts] If `polemethod` is set to `ESMF_POLEMETHOD_NPNTAVG`, this argument is required to specify how many points should be averaged over at the pole.

[unmappedaction] Specifies what should happen if there are destination points that can't be mapped to a source cell. Please see Section **??** for a list of valid options. If not specified, `unmappedaction` defaults to `ESMF_UNMAPPEDACTION_ERROR`.

[ignoreDegenerate] Ignore degenerate cells when checking the input Grids or Meshes for errors. If this is set to true, then the regridding proceeds, but degenerate cells will be skipped. If set to false, a degenerate cell produces an error. If not specified, `ignoreDegenerate` defaults to false.

[srcRegionalFlag] If .TRUE., the source grid is a regional grid, otherwise, it is a global grid. The default value is .FALSE.

[dstRegionalFlag] If .TRUE., the destination grid is a regional grid, otherwise, it is a global grid. The default value is .FALSE.

[verboseFlag] If .TRUE., it will print summary information about the regrid parameters, default to .FALSE.

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

24.4 Restrictions and Future Work

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

25 FieldBundle Class

25.1 Description

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

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

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

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

25.2 Use and Examples

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

25.2.1 Creating a FieldBundle from a list of Fields

A user can create a FieldBundle from a predefined list of Fields. In the following example, we first create an `ESMF_Grid`, then build 3 different `ESMF_Field`s with different names. The `ESMF_FieldBundle` is created from the list of 3 Fields.

```
!------------------------------------------------------------------------
!   !  Create several Fields and add them to a new FieldBundle.

    grid = ESMF_GridCreateNoPeriDim(minIndex=(/1,1/), maxIndex=(/100,200/), &
                                    regDecomp=(/2,2/), name="atmgrid", rc=rc)



    call ESMF_ArraySpecSet(arrayspec, 2, ESMF_TYPEKIND_R8, rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

    field(1) = ESMF_FieldCreate(grid, arrayspec, &
                                staggerloc=ESMF_STAGGERLOC_CENTER, &
                                name="temperature", rc=rc)
```

```
    field(2) = ESMF_FieldCreate(grid, arrayspec, &
                                staggerloc=ESMF_STAGGERLOC_CENTER, &
                                name="pressure", rc=rc)


    field(3) = ESMF_FieldCreate(grid, arrayspec, &
                                staggerloc=ESMF_STAGGERLOC_CENTER, &
                                name="heat flux", rc=rc)


    bundle1 = ESMF_FieldBundleCreate(fieldList=field(1:3), &
                                name="atmosphere data", rc=rc)

    print *, "FieldBundle example 1 returned"
```

25.2.2 Creating an empty FieldBundle then add one Field to it

A user can create an empty FieldBundle then add Fields to the empty FieldBundle. In the following example, we use the previously defined `ESMF_Grid` to build an `ESMF_Field`. An empty `ESMF_FieldBundle` is created, then the Field is added to the FieldBundle.

```
!------------------------------------------------------------------------
!   !  Create an empty FieldBundle and then add a single field to it.


    simplefield = ESMF_FieldCreate(grid, arrayspec, &
                  staggerloc=ESMF_STAGGERLOC_CENTER, name="rh", rc=rc)


    bundle2 = ESMF_FieldBundleCreate(name="time step 1", rc=rc)


    call ESMF_FieldBundleAdd(bundle2, (/simplefield/), rc=rc)


    call ESMF_FieldBundleGet(bundle2, fieldCount=fieldcount, rc=rc)

    print *, "FieldBundle example 2 returned, fieldcount =", fieldcount
```

25.2.3 Creating an empty FieldBundle then add a list of Fields to it

A user can create an empty FieldBundle then add multiple Fields to the empty FieldBundle. In the following example, we use the previously defined `ESMF_Grid` and `ESMF_Field`s. An empty `ESMF_FieldBundle` is created, then three Fields are added to the FieldBundle.

```
!------------------------------------------------------------------------
!   !  Create an empty FieldBundle and then add multiple fields to it.

    bundle3 = ESMF_FieldBundleCreate(name="southern hemisphere", rc=rc)
```

```
    call ESMF_FieldBundleAdd(bundle3, field(1:3), rc=rc)


    call ESMF_FieldBundleGet(bundle3, fieldCount=fieldcount, rc=rc)

    print *, "FieldBundle example 3 returned, fieldcount =", fieldcount
```

25.2.4 Query a Field stored in the FieldBundle by name or index

Users can query a Field stored in a FieldBundle by the Field's name or index. In the following example, the pressure Field stored in FieldBundle is queried by its name then by its index through `ESMF_FieldBundleGet()` method.

```
!------------------------------------------------------------------------
!   !  Get a Field back from a FieldBundle, first by name and then by index.
!   !  Also get the FieldBundle name.

    call ESMF_FieldBundleGet(bundle1, "pressure", field=returnedfield1, rc=rc)


    call ESMF_FieldGet(returnedfield1, name=fname1, rc=rc)


    call ESMF_FieldBundleGet(bundle1, 2, returnedfield2, rc=rc)


    call ESMF_FieldGet(returnedfield2, name=fname2, rc=rc)


    call ESMF_FieldBundleGet(bundle1, name=bname1, rc=rc)

    print *, "FieldBundle example 4 returned, field names = ", &
                  trim(fname1), ", ", trim(fname2)
    print *, "FieldBundle name = ", trim(bname1)
```

25.2.5 Query FieldBundle for Fields list either alphabetical or in order of addition

Users can query the list of Fields stored in a FieldBundle. By default the returned list of Fields are ordered alphabetically by the Field names. Users can also retrieve the list of Fields in the order by which the Fields were added to the FieldBundle.

```
    call ESMF_FieldBundleGet(bundle1, fieldList=r_fields, rc=rc)


    do i = 1, 3
      call ESMF_FieldGet(r_fields(i), name=fname1, rc=rc)
```

```
  print *, fname1
enddo


call ESMF_FieldBundleGet(bundle1, fieldList=r_fields, &
  itemorderflag=ESMF_ITEMORDER_ADDORDER, rc=rc)


do i = 1, 3
  call ESMF_FieldGet(r_fields(i), name=fname1, rc=rc)


  print *, fname1
enddo
```

25.2.6 Create a packed FieldBundle on a Grid

Create a packed fieldbundle from user supplied field names and a packed Fortran array pointer that contains the data of the packed fields on a Grid.

Create a 2D grid of 4x1 regular decomposition on 4 PETs, each PET has 10x50 elements. The index space of the entire Grid is 40x50.

```
gridxy = ESMF_GridCreateNoPeriDim(maxIndex=(/40,50/), regDecomp=(/4,1/), rc=rc)
```

Allocate a packed Fortran array pointer containing 10 packed fields, each field has 3 time slices and uses the 2D grid created. Note that gridToFieldMap uses the position of the grid dimension as elements, 3rd element of the packedPtr is 10, 4th element of the packedPtr is 50.

```
allocate(packedPtr(10, 3, 10, 50)) ! fieldDim, time, y, x
fieldDim = 1
packedFB = ESMF_FieldBundleCreate(fieldNameList, packedPtr, gridxy, fieldDim, &
gridToFieldMap=(/3,4/), staggerloc=ESMF_Staggerloc_Center, rc=rc)
```

25.2.7 Create a packed FieldBundle on a Mesh

Similarly we could create a packed fieldbundle from user supplied field names and a packed Fortran array pointer that contains the data of the packed fields on a Mesh.

Due to the verbosity of the MeshCreate process, the code for MeshCreate is not shown below, user can either refer to the MeshCreate section **??** or examine the FieldBundleCreate example source code contained in the ESMF source distribution directly. A ESMF Mesh on 4 PETs with one mesh element on each PET is created.

Allocate the packed Fortran array pointer, the first dimension is fieldDim; second dimension is the data associated with mesh element, since there is only one mesh element on each processor in this example, the allocation is 1; last dimension is the time dimension which contains 3 time slices.

```
  allocate(packedPtr3D(10, 1, 3))
```

```
    fieldDim = 1
    packedFB = ESMF_FieldBundleCreate(fieldNameList, packedPtr3D, meshEx, fieldDim, &
      gridToFieldMap=(/2/), meshloc=ESMF_MESHLOC_ELEMENT, rc=rc)
```

25.2.8 Destroy a FieldBundle

The user must call `ESMF_FieldBundleDestroy()` before deleting any of the Fields it contains. Because Fields can be shared by multiple FieldBundles and States, they are not deleted by this call.

```
!------------------------------------------------------------------------

     call ESMF_FieldBundleDestroy(bundle1, rc=rc)
```

25.2.9 Redistribute data from a source FieldBundle to a destination FieldBundle

The `ESMF_FieldBundleRedist` interface can be used to redistribute data from source FieldBundle to destination FieldBundle. This interface is overloaded by type and kind; In the version of `ESMF_FieldBundleRedist` without factor argument, a default value of factor 1 is used.

In this example, we first create two FieldBundles, a source FieldBundle and a destination FieldBundle. Then we use `ESMF_FieldBundleRedist` to redistribute data from source FieldBundle to destination FieldBundle.

```
    ! perform redist
    call ESMF_FieldBundleRedistStore(srcFieldBundle, dstFieldBundle, &
         routehandle, rc=rc)


    call ESMF_FieldBundleRedist(srcFieldBundle, dstFieldBundle, &
         routehandle, rc=rc)
```

25.2.10 Redistribute data from a packed source FieldBundle to a packed destination FieldBundle

The `ESMF_FieldBundleRedist` interface can be used to redistribute data from source FieldBundle to destination FieldBundle when both Bundles are packed with same number of fields.

In this example, we first create two packed FieldBundles, a source FieldBundle and a destination FieldBundle. Then we use `ESMF_FieldBundleRedist` to redistribute data from source FieldBundle to destination FieldBundle.

The same Grid is used where the source and destination packed FieldBundle are built upon. Source and destination Bundle have different memory layout.

```
    allocate(srcfptr(3,5,10), dstfptr(10,5,3))
    srcfptr = lpe
    srcFieldBundle = ESMF_FieldBundleCreate((/'field01', 'field02', 'field03'/), &
      srcfptr, grid, 1, gridToFieldMap=(/2,3/), rc=rc)
```

```
dstFieldBundle = ESMF_FieldBundleCreate((/'field01', 'field02', 'field03'/), &
  dstfptr, grid, 3, gridToFieldMap=(/2,1/), rc=rc)


! perform redist
call ESMF_FieldBundleRedistStore(srcFieldBundle, dstFieldBundle, &
     routehandle, rc=rc)


call ESMF_FieldBundleRedist(srcFieldBundle, dstFieldBundle, &
     routehandle, rc=rc)
```

25.2.11 Perform sparse matrix multiplication from a source FieldBundle to a destination FieldBundle

The `ESMF_FieldBundleSMM` interface can be used to perform SMM from source FieldBundle to destination FieldBundle. This interface is overloaded by type and kind;

In this example, we first create two FieldBundles, a source FieldBundle and a destination FieldBundle. Then we use `ESMF_FieldBundleSMM` to perform sparse matrix multiplication from source FieldBundle to destination FieldBundle.

The operation performed in this example is better illustrated in section 26.3.33.

Section 28.2.18 provides a detailed discussion of the sparse matrix multiplication operation implemented in ESMF.

```
call ESMF_VMGetCurrent(vm, rc=rc)


call ESMF_VMGet(vm, localPet=lpe, rc=rc)


! create distgrid and grid
distgrid = ESMF_DistGridCreate(minIndex=(/1/), maxIndex=(/16/), &
    regDecomp=(/4/), &
    rc=rc)


grid = ESMF_GridCreate(distgrid=distgrid, &
    gridEdgeLWidth=(/0/), gridEdgeUWidth=(/0/), &
    name="grid", rc=rc)


call ESMF_ArraySpecSet(arrayspec, 1, ESMF_TYPEKIND_I4, rc=rc)


! create field bundles and fields
srcFieldBundle = ESMF_FieldBundleCreate(rc=rc)


dstFieldBundle = ESMF_FieldBundleCreate(rc=rc)
```

```
do i = 1, 3
    srcField(i) = ESMF_FieldCreate(grid, arrayspec, &
        totalLWidth=(/1/), totalUWidth=(/2/), &
        rc=rc)


    call ESMF_FieldGet(srcField(i), localDe=0, farrayPtr=srcfptr, rc=rc)


    srcfptr = 1

    call ESMF_FieldBundleAdd(srcFieldBundle, (/srcField(i)/), rc=rc)


    dstField(i) = ESMF_FieldCreate(grid, arrayspec, &
        totalLWidth=(/1/), totalUWidth=(/2/), &
        rc=rc)


    call ESMF_FieldGet(dstField(i), localDe=0, farrayPtr=dstfptr, rc=rc)


    dstfptr = 0

    call ESMF_FieldBundleAdd(dstFieldBundle, (/dstField(i)/), rc=rc)


enddo

! initialize factorList and factorIndexList
allocate(factorList(4))
allocate(factorIndexList(2,4))
factorList = (/1,2,3,4/)
factorIndexList(1,:) = (/lpe*4+1,lpe*4+2,lpe*4+3,lpe*4+4/)
factorIndexList(2,:) = (/lpe*4+1,lpe*4+2,lpe*4+3,lpe*4+4/)
call ESMF_FieldBundleSMMStore(srcFieldBundle, dstFieldBundle, &
    routehandle, factorList, factorIndexList, rc=rc)


! perform smm
call ESMF_FieldBundleSMM(srcFieldBundle, dstFieldBundle, routehandle, &
      rc=rc)


! release SMM route handle
call ESMF_FieldBundleSMMRelease(routehandle, rc=rc)
```

25.2.12 Perform FieldBundle halo update

`ESMF_FieldBundleHalo` interface can be used to perform halo updates for all the Fields contained in the `ESMF_FieldBundle`.

In this example, we will set up a FieldBundle for a 2D inviscid and compressible flow problem. We will illustrate the FieldBundle halo update operation but we will not solve the non-linear PDEs. The emphasis here is to demonstrate how to set up halo regions, how a numerical scheme updates the exclusive regions, and how a halo update communicates data in the halo regions. Here are the governing equations:

$u_t + uu_x + vu_y + \frac{1}{\rho}p_x = 0$ (conservation of momentum in x-direction)

$v_t + uv_x + vv_y + \frac{1}{\rho}p_y = 0$ (conservation of momentum in y-direction)

$\rho_t + \rho u_x + \rho v_y = 0$ (conservation of mass)

$\frac{\rho}{\rho^\gamma} + u(\frac{p}{\rho^\gamma})_x + v(\frac{p}{\rho^\gamma})_y = 0$ (conservation of energy)

The four unknowns are pressure p, density ρ, velocity (u, v). The grids are set up using Arakawa D stagger (p on corner, ρ at center, u and v on edges). p, ρ, u, and v are bounded by necessary boundary conditions and initial conditions.

Section 28.2.15 provides a detailed discussion of the halo operation implemented in ESMF.

```
! create distgrid and grid according to the following decomposition
! and stagger pattern, r is density.
!
! p--------u-------+p+-------u--------p
! !                |                  |
! !                |                  |
! !                |                  |
! v        r       v        r         v
! !      PET 0     |      PET 1       |
! !                |                  |
! !                |                  |
! p--------u-------+p+-------u--------p
! !                |                  |
! !                |                  |
! !                |                  |
! v        r       v        r         v
! !      PET 2     |      PET 3       |
! !                |                  |
! !                |                  |
! p--------u-------+p+-------u--------p
!
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/256,256/), &
    regDecomp=(/2,2/), &
    rc=rc)


grid = ESMF_GridCreate(distgrid=distgrid, name="grid", rc=rc)


call ESMF_ArraySpecSet(arrayspec, 2, ESMF_TYPEKIND_R4, rc=rc)


! create field bundles and fields
fieldBundle = ESMF_FieldBundleCreate(rc=rc)


! set up exclusive/total region for the fields
```

```
!
! halo: L/U, nDim, nField, nPet
! halo configuration for pressure, and similarly for density, u, and v
halo(1,1,1,1) = 0
halo(2,1,1,1) = 0
halo(1,2,1,1) = 0
halo(2,2,1,1) = 0
halo(1,1,1,2) = 1   ! halo in x direction on left hand side of pet 1
halo(2,1,1,2) = 0
halo(1,2,1,2) = 0
halo(2,2,1,2) = 0
halo(1,1,1,3) = 0
halo(2,1,1,3) = 1   ! halo in y direction on upper side of pet 2
halo(1,2,1,3) = 0
halo(2,2,1,3) = 0
halo(1,1,1,4) = 1   ! halo in x direction on left hand side of pet 3
halo(2,1,1,4) = 1   ! halo in y direction on upper side of pet 3
halo(1,2,1,4) = 0
halo(2,2,1,4) = 0


! names and staggers of the 4 unknown fields
names(1) = "pressure"
names(2) = "density"
names(3) = "u"
names(4) = "v"
staggers(1) = ESMF_STAGGERLOC_CORNER
staggers(2) = ESMF_STAGGERLOC_CENTER
staggers(3) = ESMF_STAGGERLOC_EDGE2
staggers(4) = ESMF_STAGGERLOC_EDGE1

! create a FieldBundle
lpe = lpe + 1
do i = 1, 4
    field(i) = ESMF_FieldCreate(grid, arrayspec, &
            totalLWidth=(/halo(1,1,i,lpe), halo(1,2,i,lpe)/), &
            totalUWidth=(/halo(2,1,i,lpe), halo(2,2,i,lpe)/), &
            staggerloc=staggers(i), name=names(i), &
            rc=rc)


    call ESMF_FieldBundleAdd(fieldBundle, (/field(i)/), rc=rc)


enddo

! compute the routehandle
call ESMF_FieldBundleHaloStore(fieldBundle, routehandle=routehandle, &
                               rc=rc)


do iter = 1, 10
    do i = 1, 4
        call ESMF_FieldGet(field(i), farrayPtr=fptr, &
```

```
            exclusiveLBound=excllb, exclusiveUBound=exclub, rc=rc)


        sizes = exclub - excllb
        ! fill the total region with 0.
        fptr = 0.
        ! only update the exclusive region on local PET
        do j = excllb(1), exclub(1)
          do k = excllb(2), exclub(2)
            fptr(j,k) = iter * cos(2.*PI*j/sizes(1))*sin(2.*PI*k/sizes(2))
          enddo
        enddo
      enddo
      ! call halo execution to update the data in the halo region,
      ! it can be verified that the halo regions change from 0.
      ! to non zero values.
      call ESMF_FieldBundleHalo(fieldbundle, routehandle=routehandle, rc=rc)


  enddo
  ! release halo route handle
  call ESMF_FieldBundleHaloRelease(routehandle, rc=rc)
```

25.3 Restrictions and Future Work

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

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

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

4. **Packed data has limited supported.** One of the options that we are currently working on for FieldBundles is packing. Packing means that the data from all the Fields that comprise the FieldBundle are manipulated collectively. This operation can be done without destroying the original Field data. Packing is being designed to facilitate optimized regridding, data communication, and I/O operations. This will reduce the latency overhead of the communication.

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

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

25.4 Design and Implementation Notes

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

25.5 Class API: Basic FieldBundle Methods

25.5.1 ESMF_FieldBundleAssignment(=) - FieldBundle assignment

INTERFACE:

```
interface assignment(=)
fieldbundle1 = fieldbundle2
```

ARGUMENTS:

```
type(ESMF_FieldBundle) :: fieldbundle1
type(ESMF_FieldBundle) :: fieldbundle2
```

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:

Assign fieldbundle1 as an alias to the same ESMF fieldbundle object in memory as fieldbundle2. If fieldbundle2 is invalid, then fieldbundle1 will be equally invalid after the assignment.

The arguments are:

fieldbundle1 The `ESMF_FieldBundle` object on the left hand side of the assignment.

fieldbundle2 The `ESMF_FieldBundle` object on the right hand side of the assignment.

25.5.2 ESMF_FieldBundleOperator(==) - FieldBundle equality operator

INTERFACE:

```
interface operator(==)
if (fieldbundle1 == fieldbundle2) then ... endif
OR
result = (fieldbundle1 == fieldbundle2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_FieldBundle), intent(in) :: fieldbundle1
type(ESMF_FieldBundle), intent(in) :: fieldbundle2
```

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:

Test whether fieldbundle1 and fieldbundle2 are valid aliases to the same ESMF fieldbundle object in memory. For a more general comparison of two ESMF FieldBundles, going beyond the simple alias test, the ESMF_FieldBundleMatch() function (not yet implemented) must be used.

The arguments are:

fieldbundle1 The `ESMF_FieldBundle` object on the left hand side of the equality operation.

fieldbundle2 The `ESMF_FieldBundle` object on the right hand side of the equality operation.

25.5.3 ESMF_FieldBundleOperator(/=) - FieldBundle not equal operator

INTERFACE:

```
interface operator(/=)
if (fieldbundle1 /= fieldbundle2) then ... endif
OR
result = (fieldbundle1 /= fieldbundle2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_FieldBundle), intent(in) :: fieldbundle1
type(ESMF_FieldBundle), intent(in) :: fieldbundle2
```

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:

Test whether fieldbundle1 and fieldbundle2 are *not* valid aliases to the same ESMF fieldbundle object in memory. For a more general comparison of two ESMF FieldBundles, going beyond the simple alias test, the ESMF_FieldBundleMatch() function (not yet implemented) must be used.

The arguments are:

fieldbundle1 The `ESMF_FieldBundle` object on the left hand side of the non-equality operation.

fieldbundle2 The `ESMF_FieldBundle` object on the right hand side of the non-equality operation.

25.5.4 ESMF_FieldBundleAdd - Add Fields to a FieldBundle

INTERFACE:

```
     ! Private name; call using ESMF_FieldBundleAdd()
     subroutine ESMF_FieldBundleAddList(fieldbundle, fieldList, &
       multiflag, relaxedflag, rc)
```

ARGUMENTS:

```
     type(ESMF_FieldBundle), intent(inout) :: fieldbundle
     type(ESMF_Field), intent(in) :: fieldList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical, intent(in), optional :: multiflag
     logical, intent(in), optional :: relaxedflag
     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:

Add Field(s) to a FieldBundle. It is an error if `fieldList` contains Fields that match by name Fields already contained in `fieldbundle` when multiflag is set to `.false.` and relaxedflag is set to `.false.`.

fieldbundle `ESMF_FieldBundle` to be added to.

fieldList List of `ESMF_Field` objects to be added.

[multiflag] A setting of `.true.` allows multiple items with the same name to be added to `ESMF_FieldBundle`. For `.false.` added items must have unique names. The default setting is `.false.`.

[relaxedflag] A setting of `.true.` indicates a relaxed definition of "add" under `multiflag=.false.` mode, where it is *not* an error if `fieldList` contains items with names that are also found in `ESMF_FieldBundle`. The `ESMF_FieldBundle` is left unchanged for these items. For `.false.` this is treated as an error condition. The default setting is `.false.`.

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

25.5.5 ESMF_FieldBundleAddReplace - Conditionally add or replace Fields in a FieldBundle

INTERFACE:

```
     subroutine ESMF_FieldBundleAddReplace(fieldbundle, fieldList, rc)
```

ARGUMENTS:

```
     type(ESMF_FieldBundle), intent(inout) :: fieldbundle
     type(ESMF_Field), intent(in) :: fieldList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     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:

Fields in `fieldList` that do not match any Fields by name in `fieldbundle` are added to the FieldBundle. Fields in `fieldList` that match any Fields by name in `fieldbundle` replace those Fields.

fieldbundle `ESMF_FieldBundle` to be manipulated.

fieldList List of `ESMF_Field` objects to be added or used as replacement.

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

25.5.6 ESMF_FieldBundleCreate - Create a non packed FieldBundle from a list of Fields

INTERFACE:

```
   ! Private name; call using ESMF_FieldBundleCreate()
   function ESMF_FieldBundleCreateDefault(fieldList, &
       multiflag, relaxedflag, name, rc)
```

RETURN VALUE:

```
type(ESMF_FieldBundle) :: ESMF_FieldBundleCreateDefault
```

ARGUMENTS:

```
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_Field), intent(in), optional :: fieldList(:)
    logical, intent(in), optional :: multiflag
    logical, intent(in), optional :: relaxedflag
    character (len=*),intent(in), optional :: name
    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:

Create an `ESMF_FieldBundle` object from a list of existing Fields.

The creation of a FieldBundle leaves the bundled Fields unchanged, they remain valid individual objects. a FieldBundle is a light weight container of Field references. The actual data remains in place, there are no data movements or duplications associated with the creation of an FieldBundle.

[fieldList] List of `ESMF_Field` objects to be bundled.

[multiflag] A setting of `.true.` allows multiple items with the same name to be added to `fieldbundle`. For `.false.` added items must have unique names. The default setting is `.false.`.

[relaxedflag] A setting of `.true.` indicates a relaxed definition of "add" under `multiflag=.false.` mode, where it is *not* an error if `fieldList` contains items with names that are also found in `fieldbundle`. The `fieldbundle` is left unchanged for these items. For `.false.` this is treated as an error condition. The default setting is `.false.`.

[name] Name of the created `ESMF_FieldBundle`. A default name is generated if not specified.

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

25.5.7 ESMF_FieldBundleCreate - Create a packed FieldBundle from Fortran array pointer and Grid

INTERFACE:

```
! Private name; call using ESMF_FieldBundleCreate()
function ESMF_FieldBundleCreateGrid<rank><type><kind>(fieldNameList, &
farrayPtr, grid, fieldDim, &
indexflag, staggerLoc, &
gridToFieldMap, &
totalLWidth, totalUWidth, name, rc)
```

RETURN VALUE:

```
type(ESMF_FieldBundle) :: ESMF_FieldBundleCreateGridDataPtr<rank><type><kind>
```

ARGUMENTS:

```
character(len=*), intent(in) :: fieldNameList(:)
<type> (ESMF_KIND_<kind>), dimension(<rank>), pointer :: farrayPtr
type(ESMF_Grid), intent(in) :: grid
integer, intent(in) :: fieldDim
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_Index_Flag), intent(in), optional :: indexflag
type(ESMF_StaggerLoc), intent(in), optional :: staggerloc
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: totalLWidth(:)
integer, intent(in), optional :: totalUWidth(:)
integer, intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create a packed FieldBundle from user supplied list of field names, pre-allocated Fortran array pointer, and `ESMF_Grid` object.

The arguments are:

fieldNameList A list of field names for the Fields held by the packed `FieldBundle`.

farrayPtr Pre-allocated Fortran array pointer holding the memory of the list of Fields.

grid The `ESMF_Grid` object on which the Fields in the packed FieldBundle are built.

fieldDim The dimension in the `farrayPtr` that contains the indices of Fields to be packed.

[indexflag] Indicate how DE-local indices are defined. See section **??** for a list of valid indexflag options. All Fields in packed FieldBundle use identical indexflag setting.

[staggerloc] Stagger location of data in grid cells. For valid predefined values see section 31.2.6. To create a custom stagger location see section 31.3.25. The default value is `ESMF_STAGGERLOC_CENTER`. All Fields in packed FieldBundle use identical staggerloc setting.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `farrayPtr` by specifying the appropriate `farrayPtr` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `farrayPtr` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `farrayPtr` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `farrayPtr` dimensions less the total (distributed + undistributed) dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the farrayPtr. Permutations of the order of dimensions are handled via individual communication methods. For example, an undistributed dimension can be remapped to a distributed dimension as part of the `ESMF_ArrayRedist()` operation. All Fields in packed FieldBundle use identical gridToFieldMap setting.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the `farrayPtr`. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the `farrayPtr`. That is, for each gridded dimension the `farrayPtr` size should be max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`). All Fields in packed FieldBundle use identical totalLWidth setting.

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the `farrayPtr`. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the `farrayPtr`. That is, for each gridded dimension the `farrayPtr` size should max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`). All Fields in packed FieldBundle use identical totalUWidth setting.

[name] FieldBundle name.

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

25.5.8 ESMF_FieldBundleCreate - Create a packed FieldBundle from Fortran array pointer and Mesh

INTERFACE:

```
! Private name; call using ESMF_FieldBundleCreate()
function ESMF_FieldBundleCreateMesh<rank><type><kind>(fieldNameList, &
farrayPtr, Mesh, fieldDim, &
meshLoc, gridToFieldMap, name, rc)
```

RETURN VALUE:

```
type(ESMF_FieldBundle) :: ESMF_FieldBundleCreateMeshDataPtr<rank><type><kind>
```

ARGUMENTS:

```
character(len=*), intent(in) :: fieldNameList(:)
<type> (ESMF_KIND_<kind>), dimension(<rank>), pointer :: farrayPtr
type(ESMF_Mesh), intent(in) :: mesh
integer, intent(in) :: fieldDim
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_MeshLoc), intent(in), optional:: meshloc
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create a packed FieldBundle from user supplied list of field names, pre-allocated Fortran array pointer, and `ESMF_Mesh` object.

The arguments are:

fieldNameList A list of field names for the Fields held by the packed `FieldBundle`.

farrayPtr Pre-allocated Fortran array pointer holding the memory of the list of Fields.

mesh The `ESMF_Mesh` object on which the Fields in the packed FieldBundle are built.

fieldDim The dimension in the `farrayPtr` that contains the indices of Fields to be packed.

[meshloc] The part of the Mesh on which to build the Field. For valid predefined values see Section **??**. If not set, defaults to `ESMF_MESHLOC_NODE`.

[gridToFieldMap] List with number of elements equal to the `mesh`'s dimCount. The list elements map each dimension of the `mesh` to a dimension in the `farrayPtr` by specifying the appropriate `farrayPtr` dimension index. The default is to map all of the `mesh`'s dimensions against the lowest dimensions of the `farrayPtr` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `farrayPtr` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `farrayPtr` dimensions less the total (distributed + undistributed) dimensions in the `mesh`. Ungridded dimensions must be in the same order they are stored in the farrayPtr. Permutations of the order of dimensions are handled via individual communication methods. For example, an undistributed dimension can be remapped to a distributed dimension as part of the `ESMF_ArrayRedist()` operation. All Fields in packed FieldBundle use identical gridToFieldMap setting.

[name] FieldBundle name.

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

25.5.9 ESMF_FieldBundleDestroy - Release resources associated with a FieldBundle

INTERFACE:

```
subroutine ESMF_FieldBundleDestroy(fieldbundle, noGarbage, rc)
```

ARGUMENTS:

```
 type(ESMF_FieldBundle), intent(inout) :: fieldbundle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
 logical, intent(in), optional :: noGarbage
 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 `noGarbage`. The argument provides a mechanism to override the default garbage collection mechanism when destroying an ESMF object.

DESCRIPTION:

Destroy an ESMF_FieldBundle object. The member Fields are not touched by this operation and remain valid objects that need to be destroyed individually if necessary.

The arguments are:

fieldbundle ESMF_FieldBundle object to be destroyed.

[noGarbage] If set to .TRUE. the object will be fully destroyed and removed from the ESMF garbage collection system. Note however that under this condition ESMF cannot protect against accessing the destroyed object through dangling aliases – a situation which may lead to hard to debug application crashes.

It is generally recommended to leave the noGarbage argument set to .FALSE. (the default), and to take advantage of the ESMF garbage collection system which will prevent problems with dangling aliases or incorrect sequences of destroy calls. However this level of support requires that a small remnant of the object is kept in memory past the destroy call. This can lead to an unexpected increase in memory consumption over the course of execution in applications that use temporary ESMF objects. For situations where the repeated creation and destruction of temporary objects leads to memory issues, it is recommended to call with noGarbage set to .TRUE., fully removing the entire temporary object from memory.

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

25.5.10 ESMF_FieldBundleGet - Get object-wide information from a FieldBundle

INTERFACE:

```
! Private name; call using ESMF_FieldBundleGet()
subroutine ESMF_FieldBundleGetListAll(fieldbundle, &
  itemorderflag, geomtype, grid, locstream, mesh, xgrid, &
  fieldCount, fieldList, fieldNameList, isPacked, name, rc)
```

ARGUMENTS:

```
type(ESMF_FieldBundle), intent(in) :: fieldbundle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_ItemOrder_Flag), intent(in), optional :: itemorderflag
type(ESMF_GeomType_Flag), intent(out), optional :: geomtype
type(ESMF_Grid), intent(out), optional :: grid
type(ESMF_LocStream), intent(out), optional :: locstream
type(ESMF_Mesh), intent(out), optional :: mesh
type(ESMF_XGrid), intent(out), optional :: xgrid
integer, intent(out), optional :: fieldCount
type(ESMF_Field), intent(out), optional :: fieldList(:)
character(len=*), intent(out), optional :: fieldNameList(:)
logical, intent(out), optional :: isPacked
character(len=*), intent(out), optional :: name
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:

  6.1.0 Added argument `itemorderflag`. The new argument gives the user control over the order in which the items are returned.

  8.0.0 Added argument `isPacked`. The new argument allows the user to query if this is a packed FieldBundle.

DESCRIPTION:

Get the list of all Fields and field names bundled in a FieldBundle.

fieldbundle `ESMF_FieldBundle` to be queried.

[itemorderflag] Specifies the order of the returned items in the `fieldList` or the `fieldNameList`. The default is `ESMF_ITEMORDER_ABC`. See **??** for a full list of options.

[geomtype] Flag that indicates what type of geometry this FieldBundle object holds. Can be `ESMF_GEOMTYPE_GRID`, `ESMF_GEOMTYPE_MESH`, `ESMF_GEOMTYPE_LOCSTREAM`, `ESMF_GEOMTYPE_XGRID`

[grid] The Grid object that this FieldBundle object holds.

[locstream] The LocStream object that this FieldBundle object holds.

[mesh] The Mesh object that this FieldBundle object holds.

[xgrid] The XGrid object that this FieldBundle object holds.

[fieldCount] Upon return holds the number of Fields bundled in the fieldbundle.

[fieldList] Upon return holds a list of Fields bundled in `ESMF_FieldBundle`. The argument must be allocated to be at least of size `fieldCount`.

[fieldNameList] Upon return holds a list of the names of the fields bundled in `ESMF_FieldBundle`. The argument must be allocated to be at least of size `fieldCount`.

[isPacked] Upon return holds the information if this FieldBundle is packed.

[name] Name of the fieldbundle object.

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

25.5.11 ESMF_FieldBundleGet - Get information about a Field by name and optionally return a Field

INTERFACE:

```
    ! Private name; call using ESMF_FieldBundleGet()
    subroutine ESMF_FieldBundleGetItem(fieldbundle, fieldName, &
      field, fieldCount, isPresent, rc)
```

ARGUMENTS:

```
    type(ESMF_FieldBundle), intent(in) :: fieldbundle
    character(len=*), intent(in) :: fieldName
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_Field), intent(out), optional :: field
    integer, intent(out), optional :: fieldCount
    logical, intent(out), optional :: isPresent
    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:

Get information about items that match `fieldName` in FieldBundle.

fieldbundle `ESMF_FieldBundle` to be queried.

fieldName Specified name.

[field] Upon return holds the requested field item. It is an error if this argument was specified and there is not exactly one field item in `ESMF_FieldBundle` that matches `fieldName`.

[fieldCount] Number of Fields with `fieldName` in `ESMF_FieldBundle`.

[isPresent] Upon return indicates whether field(s) with `fieldName` exist in `ESMF_FieldBundle`.

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

25.5.12 ESMF_FieldBundleGet - Get a list of Fields by name

INTERFACE:

```
    ! Private name; call using ESMF_FieldBundleGet()
    subroutine ESMF_FieldBundleGetList(fieldbundle, fieldName, fieldList, &
      itemorderflag, rc)
```

ARGUMENTS:

```
    type(ESMF_FieldBundle), intent(in) :: fieldbundle
    character(len=*), intent(in) :: fieldName
    type(ESMF_Field), intent(out) :: fieldList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_ItemOrder_Flag), intent(in), optional :: itemorderflag
    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:

  6.1.0 Added argument `itemorderflag`. The new argument gives the user control over the order in which the items are returned.

DESCRIPTION:

Get the list of Fields from fieldbundle that match fieldName.

fieldbundle `ESMF_FieldBundle` to be queried.

fieldName Specified name.

fieldList List of Fields in `ESMF_FieldBundle` that match `fieldName`. The argument must be allocated to be at least of size `fieldCount` returned for this `fieldName`.

[itemorderflag] Specifies the order of the returned items in the `fieldList`. The default is `ESMF_ITEMORDER_ABC`. See **??** for a full list of options.

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

25.5.13 ESMF_FieldBundleGet - Get Fortran array pointer from a packed FieldBundle

INTERFACE:

```
! Private name; call using ESMF_FieldBundleGet()
function ESMF_FieldBundleGetDataPtr<rank><type><kind>(fieldBundle, &
localDe, farrayPtr, &
rc)
```

RETURN VALUE:

```
type(ESMF_FieldBundle) :: ESMF_FieldBundleGetDataPtr<rank><type><kind>
```

ARGUMENTS:

```
  type(ESMF_FieldBundle), intent(in) :: fieldBundle
  integer, intent(in), optional :: localDe
  <type> (ESMF_KIND_<kind>), dimension(<rank>), pointer :: farrayPtr
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  integer, intent(out), optional :: rc
```

DESCRIPTION:

Get a Fortran pointer to DE-local memory allocation within packed FieldBundle. It's erroneous to perform this call on a FieldBundle that's not packed.

The arguments are:

fieldBundle `ESMF_FieldBundle` object.

[localDe] Local DE for which information is requested. [0,..,localDeCount-1]. For localDeCount==1 the localDe argument may be omitted, in which case it will default to localDe=0. In the case where packed FieldBundle is created on a Grid, the number of localDes can be queried from the Grid attached to the FieldBundle. In the case where packed FieldBundle is created on a Mesh, the number of localDes is 1.

farrayPtr Fortran array pointer which will be pointed at DE-local memory allocation in packed FieldBundle.

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

25.5.14 ESMF_FieldBundleHalo - Execute a FieldBundle halo operation

INTERFACE:

```
  subroutine ESMF_FieldBundleHalo(fieldbundle, routehandle, &
    checkflag, rc)
```

ARGUMENTS:

```
        type(ESMF_FieldBundle), intent(inout) :: fieldbundle
        type(ESMF_RouteHandle), intent(inout) :: routehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
        logical, intent(in), optional :: checkflag
        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:

Execute a precomputed halo operation for the Fields in `fieldbundle`. The FieldBundle must match the respective FieldBundle used during `ESMF_FieldBundleHaloStore()` in *type*, *kind*, and memory layout of the *gridded*

dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

See `ESMF_FieldBundleHaloStore()` on how to precompute `routehandle`.

fieldbundle `ESMF_FieldBundle` with source data. The data in this FieldBundle may be destroyed by this call.

routehandle Handle to the precomputed Route.

[checkflag] If set to `.TRUE.` the input FieldBundle pair will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

25.5.15 ESMF_FieldBundleHaloRelease - Release resources associated with a FieldBundle halo operation

INTERFACE:

```
subroutine ESMF_FieldBundleHaloRelease(routehandle, &
  noGarbage, rc)
```

ARGUMENTS:

```
        type(ESMF_RouteHandle), intent(inout) :: routehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
        logical, intent(in), optional :: noGarbage
        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:

  8.0.0 Added argument `noGarbage`. The argument provides a mechanism to override the default garbage collection mechanism when destroying an ESMF object.

DESCRIPTION:

Release resources associated with a FieldBundle halo operation. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

[noGarbage] If set to `.TRUE.` the object will be fully destroyed and removed from the ESMF garbage collection system. Note however that under this condition ESMF cannot protect against accessing the destroyed object through dangling aliases – a situation which may lead to hard to debug application crashes.

It is generally recommended to leave the `noGarbage` argument set to `.FALSE.` (the default), and to take advantage of the ESMF garbage collection system which will prevent problems with dangling aliases or incorrect sequences of destroy calls. However this level of support requires that a small remnant of the object is kept in memory past the destroy call. This can lead to an unexpected increase in memory consumption over the course of execution in applications that use temporary ESMF objects. For situations where the repeated creation and destruction of temporary objects leads to memory issues, it is recommended to call with `noGarbage` set to `.TRUE.`, fully removing the entire temporary object from memory.

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

25.5.16 ESMF_FieldBundleHaloStore - Precompute a FieldBundle halo operation

INTERFACE:

```
subroutine ESMF_FieldBundleHaloStore(fieldbundle, routehandle, &
  rc)
```

ARGUMENTS:

```
type(ESMF_FieldBundle), intent(inout) :: fieldbundle
type(ESMF_RouteHandle), intent(inout) :: routehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
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:

Store a FieldBundle halo operation over the data in `fieldbundle`. By definition, all elements in the total Field regions that lie outside the exclusive regions will be considered potential destination elements for the halo operation. However, only those elements that have a corresponding halo source element, i.e. an exclusive element on one of the DEs, will be updated under the halo operation. Elements that have no associated source remain unchanged under halo.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_FieldBundleHalo()` on any pair of FieldBundles that matches `srcFieldBundle` and `dstFieldBundle` in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This call is *collective* across the current VM.

fieldbundle `ESMF_FieldBundle` containing data to be haloed. The data in this FieldBundle may be destroyed by this call.

routehandle Handle to the precomputed Route.

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

25.5.17 ESMF_FieldBundleIsCreated - Check whether a FieldBundle object has been created

INTERFACE:

```
function ESMF_FieldBundleIsCreated(fieldbundle, rc)
```

RETURN VALUE:

```
logical :: ESMF_FieldBundleIsCreated
```

ARGUMENTS:

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

DESCRIPTION:

Return `.true.` if the `fieldbundle` has been created. Otherwise return `.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:

fieldbundle `ESMF_FieldBundle` queried.

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

25.5.18 ESMF_FieldBundlePrint - Print FieldBundle information

INTERFACE:

```
subroutine ESMF_FieldBundlePrint(fieldbundle, rc)
```

ARGUMENTS:

```
type(ESMF_FieldBundle), intent(in) :: fieldbundle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
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:

Print internal information of the specified `fieldbundle` object.

The arguments are:

fieldbundle `ESMF_FieldBundle` object.

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

25.5.19 ESMF_FieldBundleRead - Read Fields to a FieldBundle from file(s)

INTERFACE:

```
   subroutine ESMF_FieldBundleRead(fieldbundle, fileName, &
     singleFile, timeslice, iofmt, rc)
```

ARGUMENTS:

```
     type(ESMF_FieldBundle), intent(inout) :: fieldbundle
     character(*), intent(in) :: fileName
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical, intent(in), optional :: singleFile
     integer, intent(in), optional :: timeslice
     type(ESMF_IOFmt_Flag), intent(in), optional :: iofmt
     integer, intent(out), optional :: rc
```

DESCRIPTION:

Read field data to a FieldBundle object from file(s). For this API to be functional, the environment variable `ESMF_PIO` should be set to "internal" when the ESMF library is built. Please see the section on Data I/O, **??**.

Limitations:

- Only single tile Arrays within Fields are supported.
- Not supported in `ESMF_COMM=mpiuni` mode.

The arguments are:

fieldbundle An `ESMF_FieldBundle` object.

fileName The name of the file from which fieldbundle data is read.

[singleFile] A logical flag, the default is .true., i.e., all Fields in the bundle are stored in one single file. If .false., each field is stored in separate files; these files are numbered with the name based on the argument "file". That is, a set of files are named: [file_name]001, [file_name]002, [file_name]003,...

[timeslice] The time-slice number of the variable read from file.

[iofmt] The I/O format. Please see Section **??** for the list of options. If not present, file names with a `.bin` extension will use `ESMF_IOFMT_BIN`, and file names with a `.nc` extension will use `ESMF_IOFMT_NETCDF`. Other files default to `ESMF_IOFMT_NETCDF`.

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

25.5.20 ESMF_FieldBundleRedist - Execute a FieldBundle redistribution

INTERFACE:

```
   subroutine ESMF_FieldBundleRedist(srcFieldBundle, dstFieldBundle, &
     routehandle, checkflag, rc)
```

ARGUMENTS:

```
         type(ESMF_FieldBundle), intent(in), optional :: srcFieldBundle
         type(ESMF_FieldBundle), intent(inout), optional :: dstFieldBundle
         type(ESMF_RouteHandle), intent(inout) :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
         logical, intent(in), optional :: checkflag
         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:

Execute a precomputed redistribution from `srcFieldBundle` to `dstFieldBundle`. Both `srcFieldBundle` and `dstFieldBundle` must match the respective FieldBundles used during `ESMF_FieldBundleRedistStore()` in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

The `srcFieldBundle` and `dstFieldBundle` arguments are optional in support of the situation where `srcFieldBundle` and/or `dstFieldBundle` are not defined on all PETs. The `srcFieldBundle` and `dstFieldBundle` must be specified on those PETs that hold source or destination DEs, respectively, but may be omitted on all other PETs. PETs that hold neither source nor destination DEs may omit both arguments.

It is erroneous to specify the identical FieldBundle object for `srcFieldBundle` and `dstFieldBundle` arguments.

See `ESMF_FieldBundleRedistStore()` on how to precompute `routehandle`.

This call is *collective* across the current VM.

For examples and associated documentation regarding this method see Section 25.2.9.

[srcFieldBundle] `ESMF_FieldBundle` with source data.

[dstFieldBundle] `ESMF_FieldBundle` with destination data.

routehandle Handle to the precomputed Route.

[checkflag] If set to `.TRUE.` the input FieldBundle pair will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

25.5.21 ESMF_FieldBundleRedistRelease - Release resources associated with a FieldBundle redistribution

INTERFACE:

```
subroutine ESMF_FieldBundleRedistRelease(routehandle, &
  noGarbage, rc)
```

ARGUMENTS:

```
        type(ESMF_RouteHandle), intent(inout) :: routehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
        logical, intent(in), optional :: noGarbage
        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:

  8.0.0 Added argument `noGarbage`. The argument provides a mechanism to override the default garbage collection mechanism when destroying an ESMF object.

DESCRIPTION:

Release resources associated with a FieldBundle redistribution. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

[noGarbage] If set to `.TRUE.` the object will be fully destroyed and removed from the ESMF garbage collection system. Note however that under this condition ESMF cannot protect against accessing the destroyed object through dangling aliases – a situation which may lead to hard to debug application crashes.

It is generally recommended to leave the `noGarbage` argument set to `.FALSE.` (the default), and to take advantage of the ESMF garbage collection system which will prevent problems with dangling aliases or incorrect sequences of destroy calls. However this level of support requires that a small remnant of the object is kept in memory past the destroy call. This can lead to an unexpected increase in memory consumption over the course of execution in applications that use temporary ESMF objects. For situations where the repeated creation and destruction of temporary objects leads to memory issues, it is recommended to call with `noGarbage` set to `.TRUE.`, fully removing the entire temporary object from memory.

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

25.5.22 ESMF_FieldBundleRedistStore - Precompute a FieldBundle redistribution with local factor argument

INTERFACE:

```
! Private name; call using ESMF_FieldBundleRedistStore()
subroutine ESMF_FieldBundleRedistStore<type><kind>(srcFieldBundle, &
dstFieldBundle, routehandle, factor, &
ignoreUnmatchedIndicesFlag, srcToDstTransposeMap, rc)
```

ARGUMENTS:

```
type(ESMF_FieldBundle), intent(in) :: srcFieldBundle
type(ESMF_FieldBundle), intent(inout) :: dstFieldBundle
type(ESMF_RouteHandle), intent(inout) :: routehandle
<type>(ESMF_KIND_<kind>), intent(in) :: factor
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
logical, intent(in), optional :: ignoreUnmatchedIndicesFlag(:)
integer, intent(in), optional :: srcToDstTransposeMap(:)
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:

  8.1.0 Added argument `ignoreUnmatchedIndicesFlag` to support cases where source and destination side do not cover the exact same index space.

DESCRIPTION:

Store a FieldBundle redistribution operation from `srcFieldBundle` to `dstFieldBundle`. PETs that specify a `factor` argument must use the <type><kind> overloaded interface. Other PETs call into the interface without `factor` argument. If multiple PETs specify the `factor` argument its type and kind as well as its value must match across all PETs. If none of the PETs specifies a `factor` argument the default will be a factor of 1.

Both srcFieldBundle and dstFieldBundle are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 28.2.18 for details on the definition of *sequence indices*. Redistribution corresponds to an identity mapping of the source FieldBundle vector to the destination FieldBundle vector.

Source and destination FieldBundles may be of different <type><kind>. Further source and destination FieldBundles may differ in shape, however, the number of elements must match.

It is erroneous to specify the identical FieldBundle object for srcFieldBundle and dstFieldBundle arguments.

The routine returns an ESMF_RouteHandle that can be used to call ESMF_FieldBundleRedist() on any pair of FieldBundles that matches srcFieldBundle and dstFieldBundle in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This method is overloaded for:
ESMF_TYPEKIND_I4, ESMF_TYPEKIND_I8,
ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8.

This call is collective across the current VM.

For examples and associated documentation regarding this method see Section 25.2.9.

The arguments are:

srcFieldBundle ESMF_FieldBundle with source data.

dstFieldBundle ESMF_FieldBundle with destination data. The data in this FieldBundle may be destroyed by this call.

routehandle Handle to the precomputed Route.

factor Factor by which to multiply source data.

[ignoreUnmatchedIndicesFlag] If set to .false., the *default*, source and destination side must cover the identical index space, using precisely matching sequence indices. If set to .true., mismatching sequence indices between source and destination side are silently ignored. The size of this array argument must either be 1 or equal the number of Fields in the srcFieldBundle and dstFieldBundle arguments. In the latter case, the handling of unmatched indices is specified for each Field pair separately. If only one element is specified, it is used for *all* Field pairs.

[srcToDstTransposeMap] List with as many entries as there are dimensions in srcFieldBundle. Each entry maps the corresponding srcFieldBundle dimension against the specified dstFieldBundle dimension. Mixing of distributed and undistributed dimensions is supported.

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

25.5.23 ESMF_FieldBundleRedistStore - Precompute a FieldBundle redistribution without local factor argument

INTERFACE:

```
! Private name; call using ESMF_FieldBundleRedistStore()
subroutine ESMF_FieldBundleRedistStoreNF(srcFieldBundle, dstFieldBundle, &
```

```
    routehandle, ignoreUnmatchedIndicesFlag, &
    srcToDstTransposeMap, rc)
```

ARGUMENTS:

```
    type(ESMF_FieldBundle), intent(in) :: srcFieldBundle
    type(ESMF_FieldBundle), intent(inout) :: dstFieldBundle
    type(ESMF_RouteHandle), intent(inout) :: routehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    logical, intent(in), optional :: ignoreUnmatchedIndicesFlag(:)
    integer, intent(in), optional :: srcToDstTransposeMap(:)
    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:

  8.1.0 Added argument `ignoreUnmatchedIndicesFlag` to support cases where source and destination side do not cover the exact same index space.

DESCRIPTION:

Store a FieldBundle redistribution operation from `srcFieldBundle` to `dstFieldBundle`. PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* `factorList` and `factorIndexList` arguments.

Both `srcFieldBundle` and `dstFieldBundle` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 28.2.18 for details on the definition of *sequence indices*. Redistribution corresponds to an identity mapping of the source FieldBundle vector to the destination FieldBundle vector.

Source and destination Fields may be of different <type><kind>. Further source and destination Fields may differ in shape, however, the number of elements must match.

It is erroneous to specify the identical FieldBundle object for srcFieldBundle and dstFieldBundle arguments.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_FieldBundleRedist()` on any pair of FieldBundles that matches `srcFieldBundle` and `dstFieldBundle` in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This call is collective across the current VM.

For examples and associated documentation regarding this method see Section 25.2.9.

The arguments are:

srcFieldBundle `ESMF_FieldBundle` with source data.

dstFieldBundle `ESMF_FieldBundle` with destination data. The data in this FieldBundle may be destroyed by this call.

routehandle Handle to the precomputed Route.

[ignoreUnmatchedIndicesFlag] If set to .false., the *default*, source and destination side must cover the identical index space, using precisely matching sequence indices. If set to .true., mismatching sequence indices between source and destination side are silently ignored. The size of this array argument must either be 1 or equal the number of Fields in the `srcFieldBundle` and `dstFieldBundle` arguments. In the latter case, the handling of unmatched indices is specified for each Field pair separately. If only one element is specified, it is used for *all* Field pairs.

[srcToDstTransposeMap] List with as many entries as there are dimensions in `srcFieldBundle`. Each entry maps the corresponding `srcFieldBundle` dimension against the specified `dstFieldBundle` dimension. Mixing of distributed and undistributed dimensions is supported.

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

25.5.24 ESMF_FieldBundleRegrid - Execute a FieldBundle regrid operation

INTERFACE:

```
   subroutine ESMF_FieldBundleRegrid(srcFieldBundle, dstFieldBundle, &
          routehandle, zeroregion, termorderflag, checkflag, rc)
```

ARGUMENTS:

```
         type(ESMF_FieldBundle), intent(in), optional :: srcFieldBundle
         type(ESMF_FieldBundle), intent(inout), optional :: dstFieldBundle
         type(ESMF_RouteHandle), intent(inout) :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
         type(ESMF_Region_Flag), intent(in), optional :: zeroregion
         type(ESMF_TermOrder_Flag), intent(in), optional :: termorderflag(:)
         logical, intent(in), optional :: checkflag
         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 `termorderflag`. The new argument gives the user control over the order in which the src terms are summed up.

DESCRIPTION:

Execute a precomputed regrid from `srcFieldBundle` to `dstFieldBundle`. Both `srcFieldBundle` and `dstFieldBundle` must match the respective FieldBundles used during `ESMF_FieldBundleRedistStore()` in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

The `srcFieldBundle` and `dstFieldBundle` arguments are optional in support of the situation where `srcFieldBundle` and/or `dstFieldBundle` are not defined on all PETs. The `srcFieldBundle` and `dstFieldBundle` must be specified on those PETs that hold source or destination DEs, respectively, but may be omitted on all other PETs. PETs that hold neither source nor destination DEs may omit both arguments.

It is erroneous to specify the identical FieldBundle object for `srcFieldBundle` and `dstFieldBundle` arguments.

See `ESMF_FieldBundleRegridStore()` on how to precompute `routehandle`.

This call is *collective* across the current VM.

[srcFieldBundle] `ESMF_FieldBundle` with source data.

[dstFieldBundle] `ESMF_FieldBundle` with destination data.

routehandle Handle to the precomputed Route.

[zeroregion] If set to `ESMF_REGION_TOTAL` *(default)* the total regions of all DEs in `dstFieldBundle` will be initialized to zero before updating the elements with the results of the sparse matrix multiplication. If set to `ESMF_REGION_EMPTY` the elements in `dstFieldBundle` will not be modified prior to the sparse matrix multiplication and results will be added to the incoming element values. Setting `zeroregion` to `ESMF_REGION_SELECT` will only zero out those elements in the destination FieldBundle that will be updated by the sparse matrix multiplication. See section **??** for a complete list of valid settings.

[termorderflag] Specifies the order of the source side terms in all of the destination sums. The `termorderflag` only affects the order of terms during the execution of the RouteHandle. See the **??** section for an in-depth discussion of *all* bit-for-bit reproducibility aspects related to route-based communication methods. See **??** for a full list of options. The size of this array argument must either be 1 or equal the number of Fields in the `srcFieldBundle` and `dstFieldBundle` arguments. In the latter case, the term order for each Field Regrid operation is indicated separately. If only one term order element is specified, it is used for *all* Field pairs. The default is `(/ESMF_TERMORDER_FREE/)`, allowing maximum flexibility in the order of terms for optimum performance.

[checkflag] If set to `.TRUE.` the input FieldBundle pair will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

25.5.25 ESMF_FieldBundleRegridRelease - Release resources associated with a FieldBundle regrid operation

INTERFACE:

```
subroutine ESMF_FieldBundleRegridRelease(routehandle, &
  noGarbage, rc)
```

ARGUMENTS:

```
         type(ESMF_RouteHandle), intent(inout) :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
         logical, intent(in), optional :: noGarbage
         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:

  8.0.0 Added argument `noGarbage`. The argument provides a mechanism to override the default garbage collection mechanism when destroying an ESMF object.

DESCRIPTION:

Release resources associated with a FieldBundle regrid operation. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

[noGarbage] If set to `.TRUE.` the object will be fully destroyed and removed from the ESMF garbage collection system. Note however that under this condition ESMF cannot protect against accessing the destroyed object through dangling aliases – a situation which may lead to hard to debug application crashes.

It is generally recommended to leave the `noGarbage` argument set to `.FALSE.` (the default), and to take advantage of the ESMF garbage collection system which will prevent problems with dangling aliases or incorrect sequences of destroy calls. However this level of support requires that a small remnant of the object is kept in memory past the destroy call. This can lead to an unexpected increase in memory consumption over the course of execution in applications that use temporary ESMF objects. For situations where the repeated creation and destruction of temporary objects leads to memory issues, it is recommended to call with `noGarbage` set to `.TRUE.`, fully removing the entire temporary object from memory.

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

25.5.26 ESMF_FieldBundleRegridStore - Precompute a FieldBundle regrid operation

INTERFACE:

```
subroutine ESMF_FieldBundleRegridStore(srcFieldBundle, dstFieldBundle, &
     srcMaskValues, dstMaskValues, regridmethod, polemethod, regridPoleNPnts, &
     lineType, normType, extrapMethod, extrapNumSrcPnts, extrapDistExponent, &
     extrapNumLevels, unmappedaction, ignoreDegenerate, srcTermProcessing, &
     pipelineDepth, routehandle, rc)
```

ARGUMENTS:

```
type(ESMF_FieldBundle), intent(in) :: srcFieldBundle
type(ESMF_FieldBundle), intent(inout) :: dstFieldBundle
integer(ESMF_KIND_I4), intent(in), optional :: srcMaskValues(:)
integer(ESMF_KIND_I4), intent(in), optional :: dstMaskValues(:)
type(ESMF_RegridMethod_Flag), intent(in), optional :: regridmethod
type(ESMF_PoleMethod_Flag), intent(in), optional :: polemethod
integer, intent(in), optional :: regridPoleNPnts
type(ESMF_LineType_Flag), intent(in), optional :: lineType
type(ESMF_NormType_Flag), intent(in), optional :: normType
type(ESMF_ExtrapMethod_Flag), intent(in), optional :: extrapMethod
integer, intent(in), optional :: extrapNumSrcPnts
real, intent(in), optional :: extrapDistExponent
integer, intent(in), optional :: extrapNumLevels
type(ESMF_UnmappedAction_Flag),intent(in), optional :: unmappedaction
logical, intent(in), optional :: ignoreDegenerate
integer, intent(inout), optional :: srcTermProcessing
integer, intent(inout), optional :: pipelineDepth
type(ESMF_RouteHandle), intent(inout), optional :: routehandle
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 arguments `ignoreDegenerate`, `lineType`, and `normType`. The argument `ignoreDegenerate` allows the user to skip degenerate cells in the regridding instead of stopping with an error. The argument `lineType` allows the user to control the path of the line between two points on a sphere surface. This allows the user to use their preferred line path for the calculation of distances and the shape of cells during regrid weight calculation on a sphere. The argument `normType` allows the user to control the type of normalization done during conservative weight generation.

  7.1.0r Added argument `srcTermProcessing`. Added argument `pipelineDepth`. The new arguments provide access to the tuning parameters affecting the performance and bit-for-bit behavior when applying the regridding weights.

  Added arguments `extrapMethod`, `extrapNumSrcPnts`, and `extrapDistExponent`. These three new extrapolation arguments allow the user to extrapolate destination points not mapped by the regrid method. `extrapMethod` allows the user to choose the extrapolation method. `extrapNumSrcPnts` and `extrapDistExponent` are parameters that allow the user to tune the behavior of the `ESMF_EXTRAPMETHOD_NEAREST_IDAVG` method.

  8.0.0 Added argument `extrapNumLevels`. For level based extrapolation methods (e.g. `ESMF_EXTRAPMETHOD_CREEP`) this argument allows the user to set how many levels to extrapolate. !

DESCRIPTION:

Store a FieldBundle regrid operation over the data in `srcFieldBundle` and `dstFieldBundle` pair.

The routine returns an ESMF_RouteHandle that can be used to call ESMF_FieldBundleRegrid() on any pair of FieldBundles that matches srcFieldBundle and dstFieldBundle in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This call is *collective* across the current VM.

srcFieldbundle Source ESMF_FieldBundle containing data to be regridded.

dstFieldbundle Destination ESMF_FieldBundle. The data in this FieldBundle may be overwritten by this call.

[srcMaskValues] Mask information can be set in the Grids (see 31.3.17) or Meshes (see **??**) upon which the Fields in the srcFieldbundle are built. The srcMaskValues argument specifies the values in that mask information which indicate a source point should be masked out. In other words, a location is masked if and only if the value for that location in the mask information matches one of the values listed in srcMaskValues. If srcMaskValues is not specified, no masking will occur.

[dstMaskValues] Mask information can be set in the Grids (see 31.3.17) or Meshes (see **??**) upon which the Fields in the dstFieldbundle are built. The dstMaskValues argument specifies the values in that mask information which indicate a destination point should be masked out. In other words, a location is masked if and only if the value for that location in the mask information matches one of the values listed in dstMaskValues. If dstMaskValues is not specified, no masking will occur.

[regridmethod] The type of interpolation. Please see Section **??** for a list of valid options. If not specified, defaults to ESMF_REGRIDMETHOD_BILINEAR.

[polemethod] Which type of artificial pole to construct on the source Grid for regridding. Please see Section **??** for a list of valid options. If not specified, defaults to ESMF_POLEMETHOD_ALLAVG.

[regridPoleNPnts] If polemethod is ESMF_POLEMETHOD_NPNTAVG. This parameter indicates how many points should be averaged over. Must be specified if polemethod is ESMF_POLEMETHOD_NPNTAVG.

[lineType] This argument controls 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. As would be expected, this argument is only applicable when srcField and dstField are built on grids which lie on the surface of a sphere. Section **??** shows a list of valid options for this argument. If not specified, the default depends on the regrid method. Section **??** has the defaults by line type. Figure 24.2.16 shows which line types are supported for each regrid method as well as showing the default line type by regrid method.

[normType] This argument controls the type of normalization used when generating conservative weights. This option only applies to weights generated with regridmethod=ESMF_REGRIDMETHOD_CONSERVE. Please see Section **??** for a list of valid options. If not specified normType defaults to ESMF_NORMTYPE_DSTAREA.

[extrapMethod] The type of extrapolation. Please see Section **??** for a list of valid options. If not specified, defaults to ESMF_EXTRAPMETHOD_NONE.

[extrapNumSrcPnts] The number of source points to use for the extrapolation methods that use more than one source point (e.g. ESMF_EXTRAPMETHOD_NEAREST_IDAVG). If not specified, defaults to 8.

[extrapDistExponent] The exponent to raise the distance to when calculating weights for the ESMF_EXTRAPMETHOD_NEAREST_IDAVG extrapolation method. A higher value reduces the influence of more distant points. If not specified, defaults to 2.0.

[extrapNumLevels] The number of levels to output for the extrapolation methods that fill levels (e.g. ESMF_EXTRAPMETHOD_CREEP). When a method is used that requires this, then an error will be returned, if it is not specified.

[unmappedaction] Specifies what should happen if there are destination points that can not be mapped to a source cell. Please see Section **??** for a list of valid options. If not specified, `unmappedaction` defaults to `ESMF_UNMAPPEDACTION_ERROR`.

[ignoreDegenerate] Ignore degenerate cells when checking the input Grids or Meshes for errors. If this is set to true, then the regridding proceeds, but degenerate cells will be skipped. If set to false, a degenerate cell produces an error. If not specified, `ignoreDegenerate` defaults to false.

[srcTermProcessing] The `srcTermProcessing` parameter controls how many source terms, located on the same PET and summing into the same destination element, are summed into partial sums on the source PET before being transferred to the destination PET. A value of 0 indicates that the entire arithmetic is done on the destination PET; source elements are neither multiplied by their factors nor added into partial sums before being sent off by the source PET. A value of 1 indicates that source elements are multiplied by their factors on the source side before being sent to the destination PET. Larger values of `srcTermProcessing` indicate the maximum number of terms in the partial sums on the source side.

Note that partial sums may lead to bit-for-bit differences in the results. See section **??** for an in-depth discussion of *all* bit-for-bit reproducibility aspects related to route-based communication methods.

The `ESMF_FieldRegridStore()` method implements an auto-tuning scheme for the `srcTermProcessing` parameter. The intent on the `srcTermProcessing` argument is "`inout`" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `srcTermProcessing` parameter, and the auto-tuning phase is skipped. In this case the `srcTermProcessing` argument is not modified on return. If the provided argument is < 0, the `srcTermProcessing` parameter is determined internally using the auto-tuning scheme. In this case the `srcTermProcessing` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `srcTermProcessing` argument is omitted.

[pipelineDepth] The `pipelineDepth` parameter controls how many messages a PET may have outstanding during a sparse matrix exchange. Larger values of `pipelineDepth` typically lead to better performance. However, on some systems too large a value may lead to performance degradation, or runtime errors.

Note that the pipeline depth has no effect on the bit-for-bit reproducibility of the results. However, it may affect the performance reproducibility of the exchange.

The `ESMF_FieldRegridStore()` method implements an auto-tuning scheme for the `pipelineDepth` parameter. The intent on the `pipelineDepth` argument is "`inout`" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `pipelineDepth` parameter, and the auto-tuning phase is skipped. In this case the `pipelineDepth` argument is not modified on return. If the provided argument is < 0, the `pipelineDepth` parameter is determined internally using the auto-tuning scheme. In this case the `pipelineDepth` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `pipelineDepth` argument is omitted.

[routehandle] Handle to the precomputed Route.

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

25.5.27 ESMF_FieldBundleRemove - Remove Fields from FieldBundle

INTERFACE:

```
subroutine ESMF_FieldBundleRemove(fieldbundle, fieldNameList, &
  multiflag, relaxedflag, rc)
```

ARGUMENTS:

```
    type(ESMF_FieldBundle), intent(inout) :: fieldbundle
    character(len=*), intent(in) :: fieldNameList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    logical, intent(in), optional :: multiflag
    logical, intent(in), optional :: relaxedflag
    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:

Remove field(s) by name from FieldBundle. In the relaxed setting it is *not* an error if `fieldNameList` contains names that are not found in `fieldbundle`.

fieldbundle `ESMF_FieldBundle` from which to remove items.

fieldNameList List of items to remove.

[multiflag] A setting of `.true.` allows multiple Fields with the same name to be removed from `fieldbundle`. For `.false.`, items to be removed must have unique names. The default setting is `.false.`.

[relaxedflag] A setting of `.true.` indicates a relaxed definition of "remove" where it is *not* an error if `fieldNameList` contains item names that are not found in `fieldbundle`. For `.false.` this is treated as an error condition. Further, in `multiflag=.false.` mode, the relaxed definition of "remove" also covers the case where there are multiple items in `fieldbundle` that match a single entry in `fieldNameList`. For `relaxedflag=.false.` this is treated as an error condition. The default setting is `.false.`.

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

25.5.28 ESMF_FieldBundleReplace - Replace Fields in FieldBundle

INTERFACE:

```
  subroutine ESMF_FieldBundleReplace(fieldbundle, fieldList, &
    multiflag, relaxedflag, rc)
```

ARGUMENTS:

```
    type(ESMF_FieldBundle), intent(inout) :: fieldbundle
    type(ESMF_Field), intent(in) :: fieldList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    logical, intent(in), optional :: multiflag
    logical, intent(in), optional :: relaxedflag
    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:

Replace field(s) by name in FieldBundle. In the relaxed setting it is not an error if `fieldList` contains Fields that do not match by name any item in `fieldbundle`. These Fields are simply ignored in this case.

fieldbundle `ESMF_FieldBundle` in which to replace items.

fieldList List of items to replace.

[multiflag] A setting of `.true.` allows multiple items with the same name to be replaced in `fieldbundle`. For `.false.`, items to be replaced must have unique names. The default setting is `.false.`.

[relaxedflag] A setting of `.true.` indicates a relaxed definition of "replace" where it is *not* an error if `fieldList` contains items with names that are not found in `fieldbundle`. These items in `fieldList` are ignored in the relaxed mode. For `.false.` this is treated as an error condition. Further, in `multiflag=.false.` mode, the relaxed definition of "replace" also covers the case where there are multiple items in `fieldbundle` that match a single entry by name in `fieldList`. For `relaxedflag=.false.` this is treated as an error condition. The default setting is `.false.`.

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

25.5.29 ESMF_FieldBundleSet - Associate a Grid with an empty FieldBundle

INTERFACE:

```
       ! Private name; call using ESMF_FieldBundleSet()
       subroutine ESMF_FieldBundleSetGrid(fieldbundle, grid, rc)
```

ARGUMENTS:

```
       type(ESMF_FieldBundle), intent(inout) :: fieldbundle
       type(ESMF_Grid), intent(in) :: grid
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       integer, intent(out), optional :: rc
```

DESCRIPTION:

Sets the `grid` for a `fieldbundle`.

The arguments are:

fieldbundle An `ESMF_FieldBundle` object.

grid The `ESMF_Grid` which all `ESMF_Field`s added to this `ESMF_FieldBundle` must have.

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

25.5.30 ESMF_FieldBundleSet - Associate a Mesh with an empty FieldBundle

INTERFACE:

```
       ! Private name; call using ESMF_FieldBundleSet()
       subroutine ESMF_FieldBundleSetMesh(fieldbundle, mesh, rc)
```

ARGUMENTS:

```
       type(ESMF_FieldBundle), intent(inout) :: fieldbundle
       type(ESMF_Mesh), intent(in) :: mesh
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       integer, intent(out), optional :: rc
```

DESCRIPTION:

Sets the `mesh` for a `fieldbundle`.

The arguments are:

fieldbundle An `ESMF_FieldBundle` object.

mesh The `ESMF_Mesh` which all `ESMF_Field`s added to this `ESMF_FieldBundle` must have.

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

25.5.31 ESMF_FieldBundleSet - Associate a LocStream with an empty FieldBundle

INTERFACE:

```
       ! Private name; call using ESMF_FieldBundleSet()
       subroutine ESMF_FieldBundleSetLS(fieldbundle, locstream, &
         rc)
```

ARGUMENTS:

```
       type(ESMF_FieldBundle), intent(inout) :: fieldbundle
       type(ESMF_LocStream), intent(in) :: locstream
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       integer, intent(out), optional :: rc
```

DESCRIPTION:

Sets the `locstream` for a `fieldbundle`.

The arguments are:

fieldbundle An ESMF_FieldBundle object.

locstream The ESMF_LocStream which all ESMF_Fields added to this ESMF_FieldBundle must have.

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

25.5.32 ESMF_FieldBundleSet - Associate a XGrid with an empty FieldBundle

INTERFACE:

```
       ! Private name; call using ESMF_FieldBundleSet()
       subroutine ESMF_FieldBundleSetXGrid(fieldbundle, xgrid, &
         rc)
```

ARGUMENTS:

```
       type(ESMF_FieldBundle), intent(inout) :: fieldbundle
       type(ESMF_XGrid), intent(in) :: xgrid
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       integer, intent(out), optional :: rc
```

DESCRIPTION:

Sets the xgrid for a fieldbundle

The arguments are:

fieldbundle An ESMF_FieldBundle object.

xgrid The ESMF_XGrid which all ESMF_Fields added to this ESMF_FieldBundle must have.

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

25.5.33 ESMF_FieldBundleSMM - Execute a FieldBundle sparse matrix multiplication

INTERFACE:

```
   subroutine ESMF_FieldBundleSMM(srcFieldBundle, dstFieldBundle, &
     routehandle, &
     zeroregion, & ! DEPRECATED ARGUMENT
     zeroregionflag, termorderflag, checkflag, rc)
```

ARGUMENTS:

```
    type(ESMF_FieldBundle), intent(in), optional :: srcFieldBundle
    type(ESMF_FieldBundle), intent(inout), optional :: dstFieldBundle
    type(ESMF_RouteHandle), intent(inout) :: routehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_Region_Flag), intent(in), optional :: zeroregion ! DEPRECATED ARGUMENT
    type(ESMF_Region_Flag), intent(in), target, optional :: zeroregionflag(:)
    type(ESMF_TermOrder_Flag), intent(in), optional :: termorderflag(:)
    logical, intent(in), optional :: checkflag
    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 `termorderflag`. The new argument gives the user control over the order in which the src terms are summed up.

  8.1.0 Added argument `zeroregionflag`, and deprecated `zeroregion`. The new argument allows greater flexibility in setting the zero region for individual FieldBundle members.

DESCRIPTION:

Execute a precomputed sparse matrix multiplication from `srcFieldBundle` to `dstFieldBundle`. Both `srcFieldBundle` and `dstFieldBundle` must match the respective FieldBundles used during `ESMF_FieldBundleRedistStore()` in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

The `srcFieldBundle` and `dstFieldBundle` arguments are optional in support of the situation where `srcFieldBundle` and/or `dstFieldBundle` are not defined on all PETs. The `srcFieldBundle` and `dstFieldBundle` must be specified on those PETs that hold source or destination DEs, respectively, but may be omitted on all other PETs. PETs that hold neither source nor destination DEs may omit both arguments.

It is erroneous to specify the identical FieldBundle object for `srcFieldBundle` and `dstFieldBundle` arguments.

See `ESMF_FieldBundleSMMStore()` on how to precompute `routehandle`.

This call is *collective* across the current VM.

For examples and associated documentation regarding this method see Section 25.2.11.

[srcFieldBundle] `ESMF_FieldBundle` with source data.

[dstFieldBundle] `ESMF_FieldBundle` with destination data.

routehandle Handle to the precomputed Route.

[zeroregion] If set to `ESMF_REGION_TOTAL` *(default)* the total regions of all DEs in all Fields in `dstFieldBundle` will be initialized to zero before updating the elements with the results of the sparse matrix multiplication. If set to `ESMF_REGION_EMPTY` the elements in the Fields in `dstFieldBundle` will not be

modified prior to the sparse matrix multiplication and results will be added to the incoming element values. Setting `zeroregion` to `ESMF_REGION_SELECT` will only zero out those elements in the destination Fields that will be updated by the sparse matrix multiplication. See section **??**

[zeroregionflag] If set to `ESMF_REGION_TOTAL` *(default)* the total regions of all DEs in the destination Field will be initialized to zero before updating the elements with the results of the sparse matrix multiplication. If set to `ESMF_REGION_EMPTY` the elements in the destination Field will not be modified prior to the sparse matrix multiplication and results will be added to the incoming element values. A setting of `ESMF_REGION_SELECT` will only zero out those elements in the destination Field that will be updated by the sparse matrix multiplication. See section **??** for a complete list of valid settings. The size of this array argument must either be 1 or equal the number of Fields in the `srcFieldBundle` and `dstFieldBundle` arguments. In the latter case, the zero region for each Field SMM operation is indicated separately. If only one zero region element is specified, it is used for *all* Field pairs.

[termorderflag] Specifies the order of the source side terms in all of the destination sums. The `termorderflag` only affects the order of terms during the execution of the RouteHandle. See the **??** section for an in-depth discussion of *all* bit-for-bit reproducibility aspects related to route-based communication methods. See **??** for a full list of options. The size of this array argument must either be 1 or equal the number of Fields in the `srcFieldBundle` and `dstFieldBundle` arguments. In the latter case, the term order for each Field SMM operation is indicated separately. If only one term order element is specified, it is used for *all* Field pairs. The default is `(/ESMF_TERMORDER_FREE/)`, allowing maximum flexibility in the order of terms for optimum performance.

[checkflag] If set to `.TRUE.` the input FieldBundle pair will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

25.5.34 ESMF_FieldBundleSMMRelease - Release resources associated with a FieldBundle sparse matrix multiplication

INTERFACE:

```
subroutine ESMF_FieldBundleSMMRelease(routehandle, &
  noGarbage, rc)
```

ARGUMENTS:

```
        type(ESMF_RouteHandle), intent(inout) :: routehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
        logical, intent(in), optional :: noGarbage
        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:

  8.0.0 Added argument `noGarbage`. The argument provides a mechanism to override the default garbage collection mechanism when destroying an ESMF object.

DESCRIPTION:

Release resources associated with a FieldBundle sparse matrix multiplication. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

[noGarbage] If set to `.TRUE.` the object will be fully destroyed and removed from the ESMF garbage collection system. Note however that under this condition ESMF cannot protect against accessing the destroyed object through dangling aliases – a situation which may lead to hard to debug application crashes.

It is generally recommended to leave the `noGarbage` argument set to `.FALSE.` (the default), and to take advantage of the ESMF garbage collection system which will prevent problems with dangling aliases or incorrect sequences of destroy calls. However this level of support requires that a small remnant of the object is kept in memory past the destroy call. This can lead to an unexpected increase in memory consumption over the course of execution in applications that use temporary ESMF objects. For situations where the repeated creation and destruction of temporary objects leads to memory issues, it is recommended to call with `noGarbage` set to `.TRUE.`, fully removing the entire temporary object from memory.

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

25.5.35 ESMF_FieldBundleSMMStore - Precompute a FieldBundle sparse matrix multiplication with local factors

INTERFACE:

```
! Private name; call using ESMF_FieldBundleSMMStore()
subroutine ESMF_FieldBundleSMMStore<type><kind>(srcFieldBundle, &
dstFieldBundle, routehandle, factorList, factorIndexList, &
ignoreUnmatchedIndicesFlag, srcTermProcessing, rc)
```

ARGUMENTS:

```
type(ESMF_FieldBundle), intent(in) :: srcFieldBundle
type(ESMF_FieldBundle), intent(inout) :: dstFieldBundle
type(ESMF_RouteHandle), intent(inout) :: routehandle
<type>(ESMF_KIND_<kind>), intent(in) :: factorList(:)
integer, intent(in), :: factorIndexList(:,:)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
logical, intent(in), optional :: ignoreUnmatchedIndicesFlag(:)
integer, intent(inout), optional :: srcTermProcessing(:)
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:

  7.1.0r Added argument `srcTermProcessing`. The new argument gives the user access to the tuning parameter affecting the sparse matrix execution and bit-wise reproducibility.

  8.1.0 Added argument `ignoreUnmatchedIndicesFlag` to support cases where the sparse matrix includes terms with source or destination sequence indices not present in the source or destination field.

DESCRIPTION:

Store a FieldBundle sparse matrix multiplication operation from `srcFieldBundle` to `dstFieldBundle`. PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* `factorList` and `factorIndexList` arguments.

Both `srcFieldBundle` and `dstFieldBundle` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 28.2.18 for details on the definition of *sequence indices*. SMM corresponds to an identity mapping of the source FieldBundle vector to the destination FieldBundle vector.

Source and destination Fields may be of different <type><kind>. Further source and destination Fields may differ in shape, however, the number of elements must match.

It is erroneous to specify the identical FieldBundle object for srcFieldBundle and dstFieldBundle arguments.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_FieldBundleSMM()` on any pair of FieldBundles that matches `srcFieldBundle` and `dstFieldBundle` in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This method is overloaded for:
`ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`,
`ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

This call is collective across the current VM.

For examples and associated documentation regarding this method see Section 25.2.11.

The arguments are:

srcFieldBundle `ESMF_FieldBundle` with source data.

dstFieldBundle `ESMF_FieldBundle` with destination data. The data in this FieldBundle may be destroyed by this call.

routehandle Handle to the precomputed Route.

factorList List of non-zero coefficients.

factorIndexList Pairs of sequence indices for the factors stored in `factorList`.

The second dimension of `factorIndexList` steps through the list of pairs, i.e. `size(factorIndexList,2) == size(factorList)`. The first dimension of `factorIndexList` is either of size 2 or size 4.

In the *size 2 format* `factorIndexList(1,:)` specifies the sequence index of the source element in the `srcFieldBundle` while `factorIndexList(2,:)` specifies the sequence index of the destination element in `dstFieldBundle`. For this format to be a valid option source and destination FieldBundles must have matching number of tensor elements (the product of the sizes of all Field tensor dimensions). Under this condition an identity matrix can be applied within the space of tensor elements for each sparse matrix factor.

The *size 4 format* is more general and does not require a matching tensor element count. Here the

`factorIndexList(1,:)` specifies the sequence index while `factorIndexList(2,:)` specifies the tensor sequence index of the source element in the `srcFieldBundle`. Further `factorIndexList(3,:)` specifies the sequence index and `factorIndexList(4,:)` specifies the tensor sequence index of the destination element in the `dstFieldBundle`.

See section 28.2.18 for details on the definition of *sequence indices* and *tensor sequence indices*.

[ignoreUnmatchedIndicesFlag] If set to .false., the *default*, source and destination side must cover all of the squence indices defined in the sparse matrix. An error will be returned if a sequence index in the sparse matrix does not match on either the source or destination side. If set to .true., mismatching sequence indices are silently ignored. The size of this array argument must either be 1 or equal the number of Fieldss in the `srcFieldBundle` and `dstFieldBundle` arguments. In the latter case, the handling of unmatched indices is specified for each Field pair separately. If only one element is specified, it is used for *all* Field pairs.

[srcTermProcessing] Source term summing options for route handle creation. See `ESMF_FieldRegridStore` documentation for a full parameter description. Two forms may be provided. If a single element list is provided, this integer value is applied across all bundle members. Otherwise, the list must contain as many elements as there are bundle members. For the special case of accessing the auto-tuned parameter (providing a negative integer value), the list length must equal the bundle member count.

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

25.5.36 ESMF_FieldBundleSMMStore - Precompute a FieldBundle sparse matrix multiplication

INTERFACE:

```
   ! Private name; call using ESMF_FieldBundleSMMStore()
     subroutine ESMF_FieldBundleSMMStoreNF(srcFieldBundle, dstFieldBundle, &
         routehandle, ignoreUnmatchedIndicesFlag, &
         srcTermProcessing, rc)
```

ARGUMENTS:

```
         type(ESMF_FieldBundle), intent(in) :: srcFieldBundle
         type(ESMF_FieldBundle), intent(inout) :: dstFieldBundle
         type(ESMF_RouteHandle), intent(inout) :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
         logical, intent(in), optional :: ignoreUnmatchedIndicesFlag(:)
         integer, intent(inout), optional :: srcTermProcessing(:)
         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:

  7.1.0r Added argument `srcTermProcessing`. The new argument gives the user access to the tuning parameter affecting the sparse matrix execution and bit-wise reproducibility.

  8.1.0 Added argument `ignoreUnmatchedIndicesFlag` to support cases where the sparse matrix includes terms with source or destination sequence indices not present in the source or destination field.

DESCRIPTION:

Store a FieldBundle sparse matrix multiplication operation from `srcFieldBundle` to `dstFieldBundle`. PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* `factorList` and `factorIndexList` arguments.

Both `srcFieldBundle` and `dstFieldBundle` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 28.2.18 for details on the definition of *sequence indices*. SMM corresponds to an identity mapping of the source FieldBundle vector to the destination FieldBundle vector.

Source and destination Fields may be of different <type><kind>. Further source and destination Fields may differ in shape, however, the number of elements must match.

It is erroneous to specify the identical FieldBundle object for srcFieldBundle and dstFieldBundle arguments.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_FieldBundleSMM()` on any pair of FieldBundles that matches `srcFieldBundle` and `dstFieldBundle` in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This method is overloaded for `ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`, `ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

This call is collective across the current VM.

For examples and associated documentation regarding this method see Section 25.2.11.

The arguments are:

srcFieldBundle `ESMF_FieldBundle` with source data.

dstFieldBundle `ESMF_FieldBundle` with destination data. The data in this FieldBundle may be destroyed by this call.

routehandle Handle to the precomputed Route.

[ignoreUnmatchedIndicesFlag] If set to .false., the *default*, source and destination side must cover all of the squence indices defined in the sparse matrix. An error will be returned if a sequence index in the sparse matrix does not

match on either the source or destination side. If set to .true., mismatching sequence indices are silently ignored. The size of this array argument must either be 1 or equal the number of Fieldss in the `srcFieldBundle` and `dstFieldBundle` arguments. In the latter case, the handling of unmatched indices is specified for each Field pair separately. If only one element is specified, it is used for *all* Field pairs.

[srcTermProcessing] Source term summing options for route handle creation. See `ESMF_FieldRegridStore` documentation for a full parameter description. Two forms may be provided. If a single element list is provided, this integer value is applied across all bundle members. Otherwise, the list must contain as many elements as there are bundle members. For the special case of accessing the auto-tuned parameter (providing a negative integer value), the list length must equal the bundle member count.

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

25.5.37 ESMF_FieldBundleSMMStore - Precompute field bundle sparse matrix multiplication using factors read from file

INTERFACE:

```
   ! Private name; call using ESMF_FieldBundleSMMStore()
     subroutine ESMF_FieldBundleSMMStoreFromFile(srcFieldBundle, dstFieldBundle, &
       filename, routehandle, ignoreUnmatchedIndicesFlag, &
       srcTermProcessing, rc)
   ! ARGUMENTS:
       type(ESMF_FieldBundle), intent(in) :: srcFieldBundle
       type(ESMF_FieldBundle), intent(inout) :: dstFieldBundle
       character(len=*), intent(in) :: filename
       type(ESMF_RouteHandle), intent(inout) :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       logical, intent(in), optional :: ignoreUnmatchedIndicesFlag(:)
       integer, intent(inout), optional :: srcTermProcessing(:)
       integer, intent(out), optional :: rc
```

DESCRIPTION:

Compute an `ESMF_RouteHandle` using factors read from file.

The arguments are:

srcFieldBundle `ESMF_FieldBundle` with source data.

dstFieldBundle `ESMF_FieldBundle` with destination data. The data in this field bundle may be destroyed by this call.

filename Path to the file containing weights for creating an `ESMF_RouteHandle`. See (12.9) for a description of the SCRIP weight file format. Only "row", "col", and "S" variables are required. They must be one-dimensionsal with dimension "n_s".

routehandle Handle to the `ESMF_RouteHandle`.

[ignoreUnmatchedIndicesFlag] If set to .false., the *default*, source and destination side must cover all of the squence indices defined in the sparse matrix. An error will be returned if a sequence index in the sparse matrix does not

match on either the source or destination side. If set to .true., mismatching sequence indices are silently ignored. The size of this array argument must either be 1 or equal the number of Fieldss in the `srcFieldBundle` and `dstFieldBundle` arguments. In the latter case, the handling of unmatched indices is specified for each Field pair separately. If only one element is specified, it is used for *all* Field pairs.

[srcTermProcessing] Source term summing options for route handle creation. See `ESMF_FieldRegridStore` documentation for a full parameter description. Two forms may be provided. If a single element list is provided, this integer value is applied across all bundle members. Otherwise, the list must contain as many elements as there are bundle members. For the special case of accessing the auto-tuned parameter (providing a negative integer value), the list length must equal the bundle member count.

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

25.5.38 ESMF_FieldBundleValidate - Validate fieldbundle internals

INTERFACE:

```
subroutine ESMF_FieldBundleValidate(fieldbundle, rc)
```

ARGUMENTS:

```
type(ESMF_FieldBundle), intent(in) :: fieldbundle
integer, intent(out), optional :: rc
```

DESCRIPTION:

Validates that the `fieldbundle` is internally consistent. The method returns an error code if problems are found.

The arguments are:

fieldbundle Specified `ESMF_FieldBundle` object.

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

25.5.39 ESMF_FieldBundleWrite - Write the Fields into a file

INTERFACE:

```
subroutine ESMF_FieldBundleWrite(fieldbundle, fileName, &
    convention, purpose, singleFile, overwrite, status, timeslice, iofmt, rc)
```

ARGUMENTS:

```
    type(ESMF_FieldBundle), intent(in) :: fieldbundle
    character(*), intent(in) :: fileName
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    character(*), intent(in), optional :: convention
    character(*), intent(in), optional :: purpose
    logical, intent(in), optional :: singleFile
    logical , intent(in), optional :: overwrite
    type(ESMF_FileStatus_Flag), intent(in), optional :: status
    integer, intent(in), optional :: timeslice
    type(ESMF_IOFmt_Flag), intent(in), optional :: iofmt
    integer, intent(out), optional :: rc
```

DESCRIPTION:

Write the Fields into a file. For this API to be functional, the environment variable `ESMF_PIO` should be set to "internal" when the ESMF library is built. Please see the section on Data I/O, **??**.

When `convention` and `purpose` arguments are specified, NetCDF dimension labels and variable attributes are written from each Field in the FieldBundle from the corresponding Attribute package. Additionally, Attributes may be set on the FieldBundle level under the same Attribute package. This allows the specification of global attributes within the file. As with individual Fields, the value associated with each name may be either a scalar character string, or a scalar or array of type integer, real, or double precision.

Limitations:

- Only single tile Fields are supported.
- Not supported in `ESMF_COMM=mpiuni` mode.

The arguments are:

fieldbundle An `ESMF_FieldBundle` object.

fileName The name of the output file to which field bundle data is written.

[convention] Specifies an Attribute package associated with the FieldBundle, and the contained Fields, used to create NetCDF dimension labels and attributes in the file. When this argument is present, the `purpose` argument must also be present. Use this argument only with a NetCDF I/O format. If binary format is used, ESMF will return an error code.

[purpose] Specifies an Attribute package associated with the FieldBundle, and the contained Fields, used to create NetCDF dimension labels and attributes in the file. When this argument is present, the `convention` argument must also be present. Use this argument only with a NetCDF I/O format. If binary format is used, ESMF will return an error code.

[singleFile] A logical flag, the default is .true., i.e., all fields in the bundle are written in one single file. If .false., each field will be written in separate files; these files are numbered with the name based on the argument "file". That is, a set of files are named: [file_name]001, [file_name]002, [file_name]003,...

[overwrite] A logical flag, the default is .false., i.e., existing field data may *not* be overwritten. If .true., the overwrite behavior depends on the value of `iofmt` as shown below:

`iofmt` = `ESMF_IOFMT_BIN`: All data in the file will be overwritten with each fields data.

`iofmt` = `ESMF_IOFMT_NETCDF, ESMF_IOFMT_NETCDF_64BIT_OFFSET`: Only the data corresponding to each fields name will be be overwritten. If the `timeslice` option is given, only data for the given timeslice may be overwritten. Note that it is always an error to attempt to overwrite a NetCDF variable with data which has a different shape.

[status] The file status. Please see Section **??** for the list of options. If not present, defaults to `ESMF_FILESTATUS_UNKNOWN`.

[timeslice] Some I/O formats (e.g. NetCDF) support the output of data in form of time slices. The `timeslice` argument provides access to this capability. `timeslice` must be positive. The behavior of this option may depend on the setting of the `overwrite` flag:

`overwrite = .false.:` If the timeslice value is less than the maximum time already in the file, the write will fail.

`overwrite = .true.:` Any positive timeslice value is valid.

By default, i.e. by omitting the `timeslice` argument, no provisions for time slicing are made in the output file, however, if the file already contains a time axis for the variable, a timeslice one greater than the maximum will be written.

[iofmt] The I/O format. Please see Section **??** for the list of options. If not present, file names with a `.bin` extension will use `ESMF_IOFMT_BIN`, and file names with a `.nc` extension will use `ESMF_IOFMT_NETCDF`. Other files default to `ESMF_IOFMT_NETCDF`.

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

26 Field Class

26.1 Description

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

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

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

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

26.1.1 Operations

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

The redistribution operation (`ESMF_FieldRedistStore()`) allows the user to move data between two Fields with the same size, but different distribution. This operation is useful, for example, to move data between two components with different distributions. Please see Section 26.3.30 for an example of the redistribution capability.

The sparse matrix multiplication operation (ESMF_FieldSMMStore()) allows the user to multiply the data in a Field by a sparse matrix. This operation is useful, for example, if the user has an interpolation matrix and wants to apply it to the data in a Field. Please see Section 26.3.33 for an example of the sparse matrix multiply capability.

The regridding operation (ESMF_FieldRegridStore()) allows the user to move data from one grid to another while maintaining certain properties of the data. Regridding is also called interpolation or remapping. In the Field regridding operation the grids the data is being moved between are the grids associated with the Fields storing the data. The regridding operation works on Fields built on Meshes, Grids, or Location Streams. There are six regridding methods available: bilinear, higher-order patch, two types of nearest neighbor, first-order conservative, and second-order conservative. Please see section 24.2 for a more indepth description of regridding including in which situations each method is supported. Please see section 26.3.25 for a description of the regridding capability as it applies to Fields. Several sections following section 26.3.25 contain examples of using regridding.

26.2 Constants

26.2.1 ESMF_FIELDSTATUS

DESCRIPTION:

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

The type of this flag is:

type(ESMF_FieldStatus_Flag)

The valid values are:

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

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

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

26.3 Use and Examples

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

The sections below go into more detail about Field usage.

26.3.1 Field create and destroy

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

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

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

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

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

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

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

26.3.2 Get Fortran data pointer, bounds, and counts information from a Field

A user can get bounds and counts information from an ESMF_Field through the ESMF_FieldGet() interface. Also available through this interface is the intrinsic Fortran data pointer contained in the internal ESMF_Array object of an ESMF_Field. The bounds and counts information are DE specific for the associated Fortran data pointer.

For a better discussion of the terminologies, bounds and widths in ESMF e.g. exclusive, computational, total bounds for the lower and upper corner of data region, etc.., user can refer to the explanation of these concepts for Grid and Array in their respective sections in the *Reference Manual*, e.g. Section 28.2.6 on Array and Section 31.3.19 on Grid.

In this example, we first create a 3D Field based on a 3D Grid and Array. Then we use the ESMF_FieldGet() interface to retrieve the data pointer, potentially updating or verifying its values. We also retrieve the bounds and counts information of the 3D Field to assist in data element iteration.

```
xdim = 180
ydim = 90
zdim = 50

! create a 3D data Field from a Grid and Array.
! first create a Grid
grid3d = ESMF_GridCreateNoPeriDim(minIndex=(/1,1,1/), &
        maxIndex=(/xdim,ydim,zdim/), &
        regDecomp=(/2,2,1/), name="grid", rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

```
call ESMF_GridGet(grid=grid3d, staggerloc=ESMF_STAGGERLOC_CENTER, &
       distgrid=distgrid3d, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

call ESMF_GridGetFieldBounds(grid=grid3d, localDe=0, &
    staggerloc=ESMF_STAGGERLOC_CENTER, totalCount=fa_shape, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

allocate(farray(fa_shape(1), fa_shape(2), fa_shape(3)) )

! create an Array
array3d = ESMF_ArrayCreate(distgrid3d, farray, &
    indexflag=ESMF_INDEX_DELOCAL, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! create a Field
field = ESMF_FieldCreate(grid=grid3d, array=array3d, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! retrieve the Fortran data pointer from the Field
call ESMF_FieldGet(field=field, localDe=0, farrayPtr=farray1, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! retrieve the Fortran data pointer from the Field and bounds
call ESMF_FieldGet(field=field, localDe=0, farrayPtr=farray1, &
    computationalLBound=compLBnd, computationalUBound=compUBnd, &
    exclusiveLBound=exclLBnd, exclusiveUBound=exclUBnd, &
    totalLBound=totalLBnd, totalUBound=totalUBnd, &
    computationalCount=comp_count, &
    exclusiveCount=excl_count, &
    totalCount=total_count, &
    rc=rc)


! iterate through the total bounds of the field data pointer
do k = totalLBnd(3), totalUBnd(3)
    do j = totalLBnd(2), totalUBnd(2)
        do i = totalLBnd(1), totalUBnd(1)
            farray1(i, j, k) = sin(2*i/total_count(1)*PI) + &
                sin(4*j/total_count(2)*PI) + &
                sin(8*k/total_count(2)*PI)
        enddo
    enddo
enddo
```

26.3.3 Get Grid, Array, and other information from a Field

A user can get the internal `ESMF_Grid` and `ESMF_Array` from a `ESMF_Field`. Note that the user should not issue any destroy command on the retrieved grid or array object since they are referenced from within the `ESMF_Field`. The retrieved objects should be used in a read-only fashion to query additional information not directly available through the `ESMF_FieldGet()` interface.

```
call ESMF_FieldGet(field, grid=grid, array=array, &
```

```
    typekind=typekind, dimCount=dimCount, staggerloc=staggerloc, &
    gridToFieldMap=gridToFieldMap, &
    ungriddedLBound=ungriddedLBound, ungriddedUBound=ungriddedUBound, &
    totalLWidth=totalLWidth, totalUWidth=totalUWidth, &
    name=name, &
    rc=rc)
```

26.3.4 Create a Field with a Grid, typekind, and rank

A user can create an ESMF_Field from an ESMF_Grid and typekind/rank. This create method associates the two objects.

We first create a Grid with a regular distribution that is 10x20 index in 2x2 DEs. This version of Field create simply associates the data with the Grid. The data is referenced explicitly on a regular 2x2 uniform grid. Finally we create a Field from the Grid, typekind, rank, and a user specified StaggerLoc.

This example also illustrates a typical use of this Field creation method. By creating a Field from a Grid and typekind/rank, the user allows the ESMF library to create a internal Array in the Field. Then the user can use ESMF_FieldGet() to retrieve the Fortran data array and necessary bounds information to assign initial values to it.

```
! create a grid
grid = ESMF_GridCreateNoPeriDim(minIndex=(/1,1/), maxIndex=(/10,20/), &
      regDecomp=(/2,2/), name="atmgrid", rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! create a Field from the Grid and arrayspec
field1 = ESMF_FieldCreate(grid, typekind=ESMF_TYPEKIND_R4, &
    indexflag=ESMF_INDEX_DELOCAL, &
    staggerloc=ESMF_STAGGERLOC_CENTER, name="pressure", rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

call ESMF_FieldGet(field1, localDe=0, farrayPtr=farray2dd, &
    totalLBound=ftlb, totalUBound=ftub, totalCount=ftc, rc=rc)

do i = ftlb(1), ftub(1)
    do j = ftlb(2), ftub(2)
        farray2dd(i, j) = sin(i/ftc(1)*PI) * cos(j/ftc(2)*PI)
    enddo
enddo

if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

26.3.5 Create a Field with a Grid and Arrayspec

A user can create an ESMF_Field from an ESMF_Grid and a ESMF_Arrayspec with corresponding rank and type. This create method associates the two objects.

We first create a Grid with a regular distribution that is 10x20 index in 2x2 DEs. This version of Field create simply associates the data with the Grid. The data is referenced explicitly on a regular 2x2 uniform grid. Then we create an ArraySpec. Finally we create a Field from the Grid, ArraySpec, and a user specified StaggerLoc.

This example also illustrates a typical use of this Field creation method. By creating a Field from a Grid and an ArraySpec, the user allows the ESMF library to create a internal Array in the Field. Then the user can use ESMF_FieldGet() to retrieve the Fortran data array and necessary bounds information to assign initial values to it.

```
! create a grid
grid = ESMF_GridCreateNoPeriDim(minIndex=(/1,1/), maxIndex=(/10,20/), &
      regDecomp=(/2,2/), name="atmgrid", rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! setup arrayspec
call ESMF_ArraySpecSet(arrayspec, 2, ESMF_TYPEKIND_R4, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! create a Field from the Grid and arrayspec
field1 = ESMF_FieldCreate(grid, arrayspec, &
     indexflag=ESMF_INDEX_DELOCAL, &
     staggerloc=ESMF_STAGGERLOC_CENTER, name="pressure", rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

call ESMF_FieldGet(field1, localDe=0, farrayPtr=farray2dd, &
    totalLBound=ftlb, totalUBound=ftub, totalCount=ftc, rc=rc)

do i = ftlb(1), ftub(1)
    do j = ftlb(2), ftub(2)
        farray2dd(i, j) = sin(i/ftc(1)*PI) * cos(j/ftc(2)*PI)
    enddo
enddo

if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

A user can also create an ArraySpec that has a different rank from the Grid, For example, the following code shows creation of of 3D Field from a 2D Grid using a 3D ArraySpec.

This example also demonstrates the technique to create a typical 3D data Field that has 2 gridded dimensions and 1 ungridded dimension.

First we create a 2D grid with an index space of 180x360 equivalent to 180x360 Grid cells (note that for a distributed memory computer, this means each grid cell will be on a separate PE!). In the FieldCreate call, we use gridToFieldMap to indicate the mapping between Grid dimension and Field dimension. For the ungridded dimension (typically the altitude), we use ungriddedLBound and ungriddedUBound to describe its bounds. Internally the ungridded dimension has a stride of 1, so the number of elements of the ungridded dimension is ungriddedUBound - ungriddedLBound + 1.

Note that gridToFieldMap in this specific example is (/1,2/) which is the default value so the user can neglect this argument for the FieldCreate call.

```
grid2d = ESMF_GridCreateNoPeriDim(minIndex=(/1,1/), &
      maxIndex=(/180,360/), regDecomp=(/2,2/), name="atmgrid", rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

call ESMF_ArraySpecSet(arrayspec, 3, ESMF_TYPEKIND_R4, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

field1 = ESMF_FieldCreate(grid2d, arrayspec, &
     indexflag=ESMF_INDEX_DELOCAL, &
```

```
     staggerloc=ESMF_STAGGERLOC_CENTER, &
     gridToFieldMap=(/1,2/), &
     ungriddedLBound=(/1/), ungriddedUBound=(/50/), &
     name="pressure", rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

26.3.6 Create a Field with a Grid and Array

A user can create an ESMF_Field from an ESMF_Grid and a ESMF_Array. The Grid was created in the previous example.

This example creates a 2D ESMF_Field from a 2D ESMF_Grid and a 2D ESMF_Array.

```
! Get necessary information from the Grid
call ESMF_GridGet(grid, staggerloc=ESMF_STAGGERLOC_CENTER, &
    distgrid=distgrid, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! Create a 2D ESMF_TYPEKIND_R4 arrayspec
call ESMF_ArraySpecSet(arrayspec, 2, ESMF_TYPEKIND_R4, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! Create a ESMF_Array from the arrayspec and distgrid
array2d = ESMF_ArrayCreate(arrayspec=arrayspec, &
        distgrid=distgrid, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! Create a ESMF_Field from the grid and array
field4 = ESMF_FieldCreate(grid, array2d, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

26.3.7 Create an empty Field and complete it with FieldEmptySet and FieldEmptyComplete

A user can create an ESMF_Field in three steps: first create an empty ESMF_Field; then set a ESMF_Grid on the empty ESMF_Field; and finally complete the ESMF_Field by calling ESMF_FieldEmptyComplete.

```
! create an empty Field
field3 = ESMF_FieldEmptyCreate(name="precip", rc=rc)


! use FieldGet to retrieve the Field Status
call ESMF_FieldGet(field3, status=fstatus, rc=rc)
```

Once the Field is created, we can verify that the status of the Field is ESMF_FIELDSTATUS_EMPTY.

```
! Test the status of the Field
if (fstatus /= ESMF_FIELDSTATUS_EMPTY) then
     call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif
```

Next we set a Grid on the empty Field. We use the 2D grid created in a previous example simply to demonstrate the method. The Field data points will be on east edge of the Grid cells with the specified ESMF_STAGGERLOC_EDGE1.

```
! Set a grid on the Field
call ESMF_FieldEmptySet(field3, grid2d, &
         staggerloc=ESMF_STAGGERLOC_EDGE1, rc=rc)


! use FieldGet to retrieve the Field Status again
call ESMF_FieldGet(field3, status=fstatus, rc=rc)


! Test the status of the Field
if (fstatus /= ESMF_FIELDSTATUS_GRIDSET) then
     call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif
```

The partially created Field is completed by specifying the typekind of its data storage. This method is overloaded with one of the following parameters, arrayspec, typekind, Fortran array, or Fortran array pointer. Additional optional arguments can be used to specify ungridded dimensions and halo regions similar to the other Field creation methods.

```
! Complete the Field by specifying the data typekind
! to be allocated internally.
call ESMF_FieldEmptyComplete(field3, typekind=ESMF_TYPEKIND_R8, &
  ungriddedLBound=(/1/), ungriddedUBound=(/5/), rc=rc)


! use FieldGet to retrieve the Field Status again
call ESMF_FieldGet(field3, status=fstatus, rc=rc)


! Test the status of the Field
if (fstatus /= ESMF_FIELDSTATUS_COMPLETE) then
     call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif
```

26.3.8 Create an empty Field and complete it with FieldEmptyComplete

A user can create an empty ESMF_Field. Then the user can finalize the empty ESMF_Field from a ESMF_Grid and an intrinsic Fortran data array. This interface is overloaded for typekind and rank of the Fortran data array.

In this example, both the grid and the Fortran array pointer are 2 dimensional and each dimension of the grid is mapped to the corresponding dimension of the Fortran array pointer, i.e. 1st dimension of grid maps to 1st dimension of Fortran array pointer, 2nd dimension of grid maps to 2nd dimension of Fortran array pointer, so on and so forth.

In order to create or complete a Field from a Grid and a Fortran array pointer, certain rules of the Fortran array bounds must be obeyed. We will discuss these rules as we progress in Field creation examples. We will make frequent reference to the terminologies for bounds and widths in ESMF. For a better discussion of these terminologies and concepts behind them, e.g. exclusive, computational, total bounds for the lower and upper corner of data region, etc.., users can refer to the explanation of these concepts for Grid and Array in their respective sections in the *Reference*

Manual, e.g. Section 28.2.6 on Array and Section 31.3.19 on Grid. The examples here are designed to help a user to get up to speed with creating Fields for typical use.

This example introduces a helper method, the `ESMF_GridGetFieldBounds` interface that facilitates the computation of Fortran data array bounds and shape to assist `ESMF_FieldEmptyComplete` finalizing a Field from an intrinsic Fortran data array and a Grid.

```
! create an empty Field
field3 = ESMF_FieldEmptyCreate(name="precip", rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! use FieldGet to retrieve total counts
call ESMF_GridGetFieldBounds(grid2d, localDe=0, &
    staggerloc=ESMF_STAGGERLOC_CENTER, totalCount=ftc, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! allocate the 2d Fortran array based on retrieved total counts
allocate(farray2d(ftc(1), ftc(2)))

! finalize the Field
call ESMF_FieldEmptyComplete(field3, grid2d, farray2d, rc=rc)
```

26.3.9 Create a 7D Field with a 5D Grid and 2D ungridded bounds from a Fortran data array

In this example, we will show how to create a 7D Field from a 5D `ESMF_Grid` and 2D ungridded bounds with arbitrary halo widths and gridToFieldMap.

We first create a 5D DistGrid and a 5D Grid based on the DistGrid; then `ESMF_GridGetFieldBounds` computes the shape of a 7D array in fsize. We can then create a 7D Field from the 5D Grid and the 7D Fortran data array with other assimilating parameters.

```
! create a 5d distgrid
distgrid5d = ESMF_DistGridCreate(minIndex=(/1,1,1,1,1/), &
    maxIndex=(/10,4,10,4,6/), regDecomp=(/2,1,2,1,1/), rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! Create a 5d Grid
grid5d = ESMF_GridCreate(distgrid=distgrid5d, name="grid", rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! use FieldGet to retrieve total counts
call ESMF_GridGetFieldBounds(grid5d, localDe=0, ungriddedLBound=(/1,2/), &
    ungriddedUBound=(/4,5/), &
    totalLWidth=(/1,1,1,2,2/), totalUWidth=(/1,2,3,4,5/), &
    gridToFieldMap=(/3,2,5,4,1/), &
    totalCount=fsize, &
    rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! allocate the 7d Fortran array based on retrieved total counts
allocate(farray7d(fsize(1), fsize(2), fsize(3), fsize(4), fsize(5), &
                  fsize(6), fsize(7)))
```

```
! create the Field
field7d = ESMF_FieldCreate(grid5d, farray7d, ESMF_INDEX_DELOCAL, &
    ungriddedLBound=(/1,2/), ungriddedUBound=(/4,5/), &
    totalLWidth=(/1,1,1,2,2/), totalUWidth=(/1,2,3,4,5/), &
    gridToFieldMap=(/3,2,5,4,1/), &
    rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

A user can allocate the Fortran array in a different manner using the lower and upper bounds returned from FieldGet through the optional totalLBound and totalUBound arguments. In the following example, we create another 7D Field by retrieving the bounds and allocate the Fortran array with this approach. In this scheme, indexing the Fortran array is sometimes more convenient than using the shape directly.

```
call ESMF_GridGetFieldBounds(grid5d, localDe=0, ungriddedLBound=(/1,2/), &
    ungriddedUBound=(/4,5/), &
    totalLWidth=(/1,1,1,2,2/), totalUWidth=(/1,2,3,4,5/), &
    gridToFieldMap=(/3,2,5,4,1/), &
    totalLBound=flbound, totalUBound=fubound, &
    rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

allocate(farray7d2(flbound(1):fubound(1), flbound(2):fubound(2), &
                   flbound(3):fubound(3), flbound(4):fubound(4), &
                   flbound(5):fubound(5), flbound(6):fubound(6), &
                   flbound(7):fubound(7)) )

field7d2 = ESMF_FieldCreate(grid5d, farray7d2, ESMF_INDEX_DELOCAL, &
    ungriddedLBound=(/1,2/), ungriddedUBound=(/4,5/), &
    totalLWidth=(/1,1,1,2,2/), totalUWidth=(/1,2,3,4,5/), &
    gridToFieldMap=(/3,2,5,4,1/), &
    rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

26.3.10 Shared memory features: DE pinning, sharing, and migration

See 28.2.13 for a introduction of the DE pinning feature. Here we focus on demonstrating the use of the DE pinning feature in the context of ESMF Field.

When an ESMF Field object is created, the specified underlying DistGrid indicates how many Decomposition Elements (DEs) are created. Each DE has its own memory allocation to hold user data. The DELayout, referenced by the DistGrid, determines which PET is considered the *owner* of each of the DEs. Queried for the local DEs, the Field object returns the list of DEs that are owned by the local PET making the query.

By default DEs are *pinned* to the PETs under which they were created. The memory allocation associated with a specific DE is only defined in the VAS of the PET to which the DE is pinned. As a consequence, only the PET owning a DE has access to its memory allocation.

On shared memory systems, however, ESMF allows DEs to be pinned to SSIs instead of PETs. In this case the PET under which a DE was created is still consider the owner, but now *all* PETs under the same SSI have access to the DE. For this the memory allocation associated with the DE is mapped into the VAS of all the PETs under the SSI.

To create an Field with each DE pinned to SSI instead of PET, first query the VM for the available level of support.

```
call ESMF_VMGet(vm, ssiSharedMemoryEnabledFlag=ssiSharedMemoryEnabled, rc=rc)
```

```
if (ssiSharedMemoryEnabled) then
```

Knowing that the SSI shared memory feature is available, it is now possible to create an Field object with DE to SSI pinning.

```
grid = ESMF_GridCreateNoPeriDim(maxIndex=(/40,10/), regDecomp=(/4,1/), &
  coordSys = ESMF_COORDSYS_CART, &
  rc=rc)


field = ESMF_FieldCreate(typekind=ESMF_TYPEKIND_R8, grid=grid, &
  pinflag=ESMF_PIN_DE_TO_SSI, rc=rc)
```

Just as in the cases discussed before, where the same Grid was used, a default DELayout with as many DEs as PETs in the VM is constructed. Setting the `pinflag` to `ESMF_PIN_DE_TO_SSI` does not change the fact that each PET owns exactly one of the DEs. However, assuming that this code is run on a set of PETs that are all located under the same SSI, every PET now has *access* to all of the DEs. The situation can be observed by querying for both the `localDeCount`, and the `ssiLocalDeCount`.

```
call ESMF_FieldGet(field, localDeCount=localDeCount, &
  ssiLocalDeCount=ssiLocalDeCount, rc=rc)
```

Assuming execution on 4 PETs, all located on the same SSI, the values of the returned variable are `localDeCount==1` and `ssiLocalDeCount==4` on all of the PETs. The mapping between each PET's local DE, and the global DE index is provided through the `localDeToDeMap` array argument. The amount of mapping information returned is dependent on how large `localDeToDeMap` has been sized by the user. For `size(localDeToDeMap)==localDeCount`, only mapping information for those DEs *owned* by the local PET is filled in. However for `size(localDeToDeMap)==ssiLocalDeCount`, mapping information for all locally *accessible* DEs is returned, including those owned by other PETs on the same SSI.

```
allocate(localDeToDeMap(0:ssiLocalDeCount-1))
call ESMF_FieldGet(field, localDeToDeMap=localDeToDeMap, rc=rc)
```

The first `localDeCount` entries of `localDeToDeMap` are always the global DE indices of the DEs *owned* by the local PET. The remaining `ssiLocalDeCount-localDeCount` entries are the global DE indices of DEs *shared* by other PETs. The ordering of the shared DEs is from smallest to greatest, excluding the locally owned DEs, which were already listed at the beginning of `localDeToDeMap`. For the current case, again assuming execution on 4 PETs all located on the same SSI, we expect the following situation:

PET 0: `localDeToDeMap==(/0,1,2,3/)`
PET 1: `localDeToDeMap==(/1,0,2,3/)`
PET 2: `localDeToDeMap==(/2,0,1,3/)`
PET 3: `localDeToDeMap==(/3,0,1,2/)`

Each PET can access the memory allocations associated with *all* of the DEs listed in the `localDeToDeMap` returned by the Field object. Direct access to the Fortran array pointer of a specific memory allocation is available through `ESMF_FieldGet()`. Here each PET queries for the `farrayPtr` of `localDe==2`, i.e. the 2nd shared DE.

```
call ESMF_FieldGet(field, farrayPtr=myFarray, localDe=2, rc=rc)
```

Now variable `myFarray` on PETs 0 and 1 both point to the *same* memory allocation for global DE 2. Both PETs have access to the same piece of shared memory! The same is true for PETs 2 and 3, pointing to the shared memory allocation of global DE 1.

It is important to note that all of the typical considerations surrounding shared memory programming apply when accessing shared DEs! Proper synchronization between PETs accessing shared DEs is critical to avoid *race conditions*. Also performance issues like *false sharing* need to be considered for optimal use.

For a simple demonstration, PETs 0 and 2 fill the entire memory allocation of DE 2 and 1, respectively, to a unique value.

```
if (localPet==0) then
  myFarray = 12345.6789d0
else if (localPet==2) then
  myFarray = 6789.12345d0
endif
```

Here synchronization is needed before any PETs that share access to the same DEs can safely access the data without race condition. The Field class provides a simple synchronization method that can be used.

```
call ESMF_FieldSync(field, rc=rc) ! prevent race condition
```

Now it is safe for PETs 1 and 3 to access the shared DEs. We expect to find the data that was set above. For simplicity of the code only the first array element is inspected here.

```
if (localPet==1) then
  if (abs(myFarray(1,1)-12345.6789d0)>1.d10) print *, "bad data detected"
else if (localPet==3) then
  if (abs(myFarray(1,1)-6789.12345d0)>1.d10) print *, "bad data detected"
endif


endif ! ending the ssiSharedMemoryEnabled conditional
```

26.3.11 Create a 2D Field with a 2D Grid and a Fortran data array

A user can create an `ESMF_Field` directly from an `ESMF_Grid` and an intrinsic Fortran data array. This interface is overloaded for typekind and rank of the Fortran data array.

In the following example, each dimension size of the Fortran array is equal to the exclusive bounds of its corresponding Grid dimension queried from the Grid through `ESMF_GridGet()` public interface.

Formally let fa_shape(i) be the shape of i-th dimension of user supplied Fortran array, then rule 1 states:

```
(1) fa_shape(i) = exclusiveCount(i)
```

```
i = 1...GridDimCount
```

fa_shape(i) defines the shape of i-th dimension of the Fortran array. ExclusiveCount are the number of data elements of i-th dimension in the exclusive region queried from `ESMF_GridGet` interface. *Rule 1 assumes that the Grid and the Fortran intrinsic array have same number of dimensions; and optional arguments of FieldCreate from Fortran array are left unspecified using default setup*. These assumptions are true for most typical uses of FieldCreate from Fortran data array. This is the easiest way to create a Field from a Grid and a Fortran intrinsic data array.

Fortran array dimension sizes (called shape in most Fortran language books) are equivalent to the bounds and counts used in this manual. The following equation holds:

```
fa_shape(i) = shape(i) = counts(i) = upper_bound(i) - lower_bound(i) + 1
```

These typically mean the same concept unless specifically explained to mean something else. For example, ESMF uses DimCount very often to mean number of dimensions instead of its meaning implied in the above equation. We'll clarify the meaning of a word when ambiguity could occur.

Rule 1 is most useful for a user working with Field creation from a Grid and a Fortran data array in most scenarios. It extends to higher dimension count, 3D, 4D, etc... Typically, as the code example demonstrates, a user first creates a Grid, then uses `ESMF_GridGet()` to retrieve the exclusive counts. Next the user calculates the shape of each Fortran array dimension according to rule 1. The Fortran data array is allocated and initialized based on the computed shape. A Field can either be created in one shot or created empty and finished using `ESMF_FieldEmptyComplete`.

There are important details that can be skipped but are good to know for `ESMF_FieldEmptyComplete` and `ESMF_FieldCreate` from a Fortran data array. 1) these methods require *each PET contains exactly one DE*. This implies that a code using FieldCreate from a data array or FieldEmptyComplete must have the same number of DEs and PETs, formally $n_{DE} = n_{PET}$. Violation of this condition will cause run time failures. 2) the bounds and counts retrieved from GridGet are DE specific or equivalently PET specific, which means that *the Fortran array shape could be different from one PET to another*.

```
grid = ESMF_GridCreateNoPeriDim(minIndex=(/1,1/), maxIndex=(/10,20/), &
      regDecomp=(/2,2/), name="atmgrid", rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

call ESMF_GridGet(grid, localDE=0, staggerloc=ESMF_STAGGERLOC_CENTER, &
    exclusiveCount=gec, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

allocate(farray(gec(1), gec(2)) )

field = ESMF_FieldCreate(grid, farray, ESMF_INDEX_DELOCAL, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

26.3.12 Create a 2D Field with a 2D Grid and a Fortran data pointer

The setup of this example is similar to the previous section except that the Field is created from a data pointer instead of a data array. We highlight the ability to deallocate the internal Fortran data pointer queried from the Field. This gives a user more flexibility with memory management.

```
allocate(farrayPtr(gec(1), gec(2)) )

field = ESMF_FieldCreate(grid, farrayPtr, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
call ESMF_FieldGet(field, farrayPtr=farrayPtr2, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
! deallocate the retrieved Fortran array pointer
deallocate(farrayPtr2)
```

26.3.13 Create a 3D Field with a 2D Grid and a 3D Fortran data array

This example demonstrates a typical use of `ESMF_Field` combining a 2D grid and a 3D Fortran native data array. One immediate problem follows: how does one define the bounds of the ungridded dimension? This is solved by the optional arguments `ungriddedLBound` and `ungriddedUBound` of the `ESMF_FieldCreate` interface. By definition, `ungriddedLBound` and `ungriddedUBound` are both 1 dimensional integer Fortran arrays.

Formally, let fa_shape(j=1...FieldDimCount-GridDimCount) be the shape of the ungridded dimensions of a Field relative to the Grid used in Field creation. The Field dimension count is equal to the number of dimensions of the Fortran array, which equals the number of dimensions of the resultant Field. GridDimCount is the number of dimensions of the Grid.

fa_shape(j) is computed as:

```
fa_shape(j) = ungriddedUBound(j) - ungriddedLBound(j) + 1
```

fa_shape is easy to compute when the gridded and ungridded dimensions do not mix. However, it's conceivable that at higher dimension count, gridded and ungridded dimensions can interleave. To aid the computation of ungridded dimension shape we formally introduce the mapping concept.

Let $map_{A,B}(i = 1...n_A) = i_B$, and $i_B \in [\phi, 1...n_B]$. n_A is the number of elements in set A, n_B is the number of elements in set B. $map_{A,B}(i)$ defines a mapping from i-th element of set A to i_B-th element in set B. $i_B = \phi$ indicates there does not exist a mapping from i-th element of set A to set B.

Suppose we have a mapping from dimension index of ungriddedLBound (or ungriddedUBound) to Fortran array dimension index, called ugb2fa. By definition, n_A equals to the dimension count of ungriddedLBound (or ungriddedUBound), n_B equals to the dimension count of the Fortran array. We can now formulate the computation of ungridded dimension shape as rule 2:

```
(2) fa_shape(ugb2fa(j)) = ungriddedUBound(j) - ungriddedLBound(j) + 1
                      j = 1..FortranArrayDimCount - GridDimCount
```

The mapping can be computed in linear time proportional to the Fortran array dimension count (or rank) using the following algorithm in pseudocode:

```
map_index = 1
do i = 1, farray_rank
    if i-th dimension of farray is ungridded
        ugb2fa(map_index) = i
```

```
        map_index = map_index + 1
    endif
enddo
```

Here we use rank and dimension count interchangeably. These 2 terminologies are typically equivalent. But there are subtle differences under certain conditions. Rank is the total number of dimensions of a tensor object. Dimension count allows a finer description of the heterogeneous dimensions in that object. For example, a Field of rank 5 can have 3 gridded dimensions and 2 ungridded dimensions. Rank is precisely the summation of dimension count of all types of dimensions.

For example, if a 5D array is used with a 3D Grid, there are 2 ungridded dimensions: ungriddedLBound=(/1,2/) and ungriddedUBound=(/5,7/). Suppose the distribution of dimensions looks like (O, X, O, X, O), O means gridded, X means ungridded. Then the mapping from ungridded bounds to Fortran array is ugb2fa=(/2, 4/). The shape of 2nd and 4th dimension of Fortran array should equal (5, 8).

Back to our 3D Field created from a 2D Grid and 3D Fortran array example, suppose the 3rd Field dimension is ungridded, ungriddedLBound=(/3/), ungriddedUBound=(/9/). First we use rule 1 to compute shapes of the gridded Fortran array dimension, then we use rule 2 to compute shapes of the ungridded Fortran array dimension. In this example, we used the exclusive bounds obtained in the previous example.

```
fa_shape(1) = gec(1) ! rule 1
fa_shape(2) = gec(2)
fa_shape(3) = 7 ! rule 2 9-3+1
allocate(farray3d(fa_shape(1), fa_shape(2), fa_shape(3)))
field = ESMF_FieldCreate(grid, farray3d, ESMF_INDEX_DELOCAL, &
    ungriddedLBound=(/3/), ungriddedUBound=(/9/), &
    rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

26.3.14 Create a 3D Field with a 2D Grid and a 3D Fortran data array with gridToFieldMap argument

Building upon the previous example, we will create a 3D Field from a 2D grid and 3D array but with a slight twist. In this example, we introduce the gridToFieldMap argument that allows a user to map Grid dimension index to Field dimension index.

In this example, both dimensions of the Grid are distributed and the mapping from DistGrid to Grid is (/1,2/). We will introduce rule 3 assuming distgridToGridMap=(/1,2,3...gridDimCount/), and distgridDimCount equals to gridDimCount. This is a reasonable assumption in typical Field use.

We apply the mapping gridToFieldMap on rule 1 to create rule 3:

```
(3) fa_shape(gridToFieldMap(i)) = exclusiveCount(i)
                                i = 1,..GridDimCount.
```

Back to our example, suppose the 2nd Field dimension is ungridded, ungriddedLBound=(/3/), ungriddedUBound=(/9/). gridToFieldMap=(/3,1/), meaning the 1st Grid dimension maps to 3rd Field dimension, and 2nd Grid dimension maps to 1st Field dimension.

First we use rule 3 to compute shapes of the gridded Fortran array dimension, then we use rule 2 to compute shapes

of the ungridded Fortran array dimension. In this example, we use the exclusive bounds obtained in the previous example.

```
gridToFieldMap2d(1) = 3
gridToFieldMap2d(2) = 1
do i = 1, 2
    fa_shape(gridToFieldMap2d(i)) = gec(i)
end do
fa_shape(2) = 7
allocate(farray3d(fa_shape(1), fa_shape(2), fa_shape(3)))
field = ESMF_FieldCreate(grid, farray3d, ESMF_INDEX_DELOCAL, &
    ungriddedLBound=(/3/), ungriddedUBound=(/9/), &
    gridToFieldMap=gridToFieldMap2d, &
    rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

26.3.15 Create a 3D Field with a 2D Grid and a 3D Fortran data array with halos

This example is similar to example 26.3.14. In addition, here we will show how a user can associate different halo widths to a Fortran array to create a Field through the totalLWidth and totalUWidth optional arguments. A diagram of the dimension configuration from Grid, halos, and Fortran data array is shown here.

The `ESMF_FieldCreate()` interface supports creating a Field from a Grid and a Fortran array padded with halos on the distributed dimensions of the Fortran array. Using this technique one can avoid passing non-contiguous Fortran array slice to FieldCreate. It guarantees the same exclusive region, and by using halos, it also defines a bigger total region to contain the entire contiguous memory block of the Fortran array.

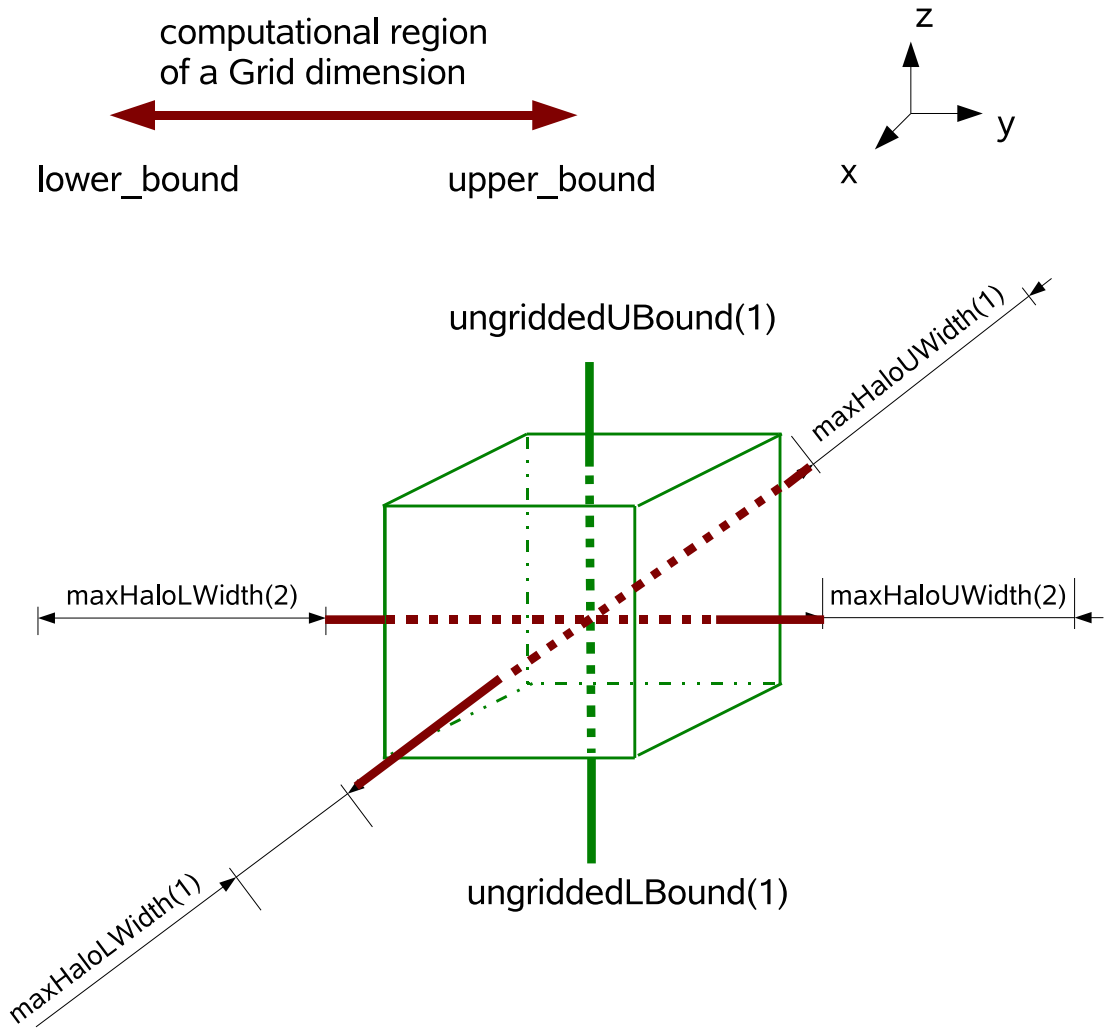
The elements of totalLWidth and totalUWidth are applied in the order distributed dimensions appear in the Fortran array. By definition, totalLWidth and totalUWidth are 1 dimensional arrays of non-negative integer values. The size of haloWidth arrays is equal to the number of distributed dimensions of the Fortran array, which is also equal to the number of distributed dimensions of the Grid used in the Field creation.

Because the order of totalWidth (representing both totalLWidth and totalUWidth) element is applied to the order distributed dimensions appear in the Fortran array dimensions, it's quite simple to compute the shape of distributed dimensions of the Fortran array. They are done in a similar manner when applying ungriddedLBound and ungriddedUBound to ungridded dimensions of the Fortran array defined by rule 2.

Assume we have the mapping from the dimension index of totalWidth to the dimension index of Fortran array, called mhw2fa; and we also have the mapping from dimension index of Fortran array to dimension index of the Grid, called fa2g. The shape of distributed dimensions of a Fortran array can be computed by rule 4:

```
(4) fa_shape(mhw2fa(k)) = exclusiveCount(fa2g(mhw2fa(k)) +
                          totalUWidth(k) + totalLWidth(k)
                    k = 1...size(totalWidth)
```

This rule may seem confusing but algorithmically the computation can be done by the following pseudocode:



ESMF_Field created from a 2D ESMF_Grid (Red) and a 3D Intrinsic Fortran data array (Green). The ungridded bounds and halo widths are applied to corresponding dimensions.

Figure 12: Field dimension configuration from Grid, halos, and Fortran data array.

```
fa_index = 1
do i = 1, farray_rank
   if i-th dimension of Fortran array is distributed
       fa_shape(i) = exclusiveCount(fa2g(i)) +
                     totalUWidth(fa_index) + totalLWidth(fa_index)
       fa_index = fa_index + 1
   endif
enddo
```

The only complication then is to figure out the mapping from Fortran array dimension index to Grid dimension index. This process can be done by computing the reverse mapping from Field to Grid.

Typically, we don't have to consider these complications if the following conditions are met: 1) All Grid dimensions are distributed. 2) DistGrid in the Grid has a dimension index mapping to the Grid in the form of natural order (/1,2,3,.../). This natural order mapping is the default mapping between various objects throughout ESMF. 3) Grid to Field mapping is in the form of natural order, i.e. default mapping. These seem like a lot of conditions but they are the default case in the interaction among DistGrid, Grid, and Field. When these conditions are met, which is typically true, the shape of distributed dimensions of Fortran array follows rule 5 in a simple form:

```
(5) fa_shape(k) = exclusiveCount(k) +
                  totalUWidth(k) + totalLWidth(k)
              k = 1...size(totalWidth)
```

Let's examine an example on how to apply rule 5. Suppose we have a 5D array and a 3D Grid that has its first 3 dimensions mapped to the first 3 dimensions of the Fortran array. totalLWidth=(/1,2,3/), totalUWidth=(/7,9,10/), then by rule 5, the following pseudo code can be used to compute the shape of the first 3 dimensions of the Fortran array. The shape of the remaining two ungridded dimensions can be computed according to rule 2.

```
do k = 1, 3
    fa_shape(k) = exclusiveCount(k) +
                  totalUWidth(k) + totalLWidth(k))
enddo
```

Suppose now gridToFieldMap=(/2,3,4/) instead which says the first dimension of Grid maps to the 2nd dimension of Field (or Fortran array) and so on and so forth, we can obtain a more general form of rule 5 by introducing first_distdim_index shift when Grid to Field map (gridToFieldMap) is in the form of (/a,a+1,a+2.../).

```
(6) fa_shape(k+first_distdim_index-1) = exclusiveCount(k) +
                                        totalUWidth(k) + totalLWidth(k)
                                  k = 1...size(totalWidth)
```

It's obvious that first_distdim_index=a. If the first dimension of the Fortran array is distributed, then rule 6 degenerates into rule 5, which is the typical case.

Back to our example creating a 3D Field from a 2D Grid and a 3D intrinsic Fortran array, we will use the Grid created from previous example that satisfies condition 1 and 2. We'll also use a simple gridToFieldMap (1,2) which is the default mapping that satisfies condition 3. First we use rule 5 to compute the shape of distributed dimensions then we use rule 2 to compute the shape of the ungridded dimensions.

```
gridToFieldMap2d(1) = 1
gridToFieldMap2d(2) = 2
totalLWidth2d(1) = 3
totalLWidth2d(2) = 4
totalUWidth2d(1) = 3
totalUWidth2d(2) = 5
do k = 1, 2
    fa_shape(k) = gec(k) + totalLWidth2d(k) + totalUWidth2d(k)
end do
fa_shape(3) = 7          ! 9-3+1
allocate(farray3d(fa_shape(1), fa_shape(2), fa_shape(3)))
field = ESMF_FieldCreate(grid, farray3d, ESMF_INDEX_DELOCAL, &
    ungriddedLBound=(/3/), ungriddedUBound=(/9/), &
    totalLWidth=totalLWidth2d, totalUWidth=totalUWidth2d, &
    gridToFieldMap=gridToFieldMap2d, &
    rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

26.3.16 Create a Field from a LocStream, typekind, and rank

In this example, an ESMF_Field is created from an ESMF_LocStream and typekind/rank. The location stream object is uniformly distributed in a 1 dimensional space on 4 DEs. The rank is 1 dimensional. Please refer to LocStream examples section for more information on LocStream creation.

```
locs = ESMF_LocStreamCreate(minIndex=1, maxIndex=16, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

field = ESMF_FieldCreate(locs, typekind=ESMF_TYPEKIND_I4, &
    rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

26.3.17 Create a Field from a LocStream and arrayspec

In this example, an ESMF_Field is created from an ESMF_LocStream and an ESMF_Arrayspec. The location stream object is uniformly distributed in a 1 dimensional space on 4 DEs. The arrayspec is 1 dimensional. Please refer to LocStream examples section for more information on LocStream creation.

```
locs = ESMF_LocStreamCreate(minIndex=1, maxIndex=16, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

call ESMF_ArraySpecSet(arrayspec, 1, ESMF_TYPEKIND_I4, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

```
field = ESMF_FieldCreate(locs, arrayspec, &
    rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

26.3.18 Create a Field from a Mesh, typekind, and rank

In this example, an ESMF_Field is created from an ESMF_Mesh and typekind/rank. The mesh object is on a Euclidean surface that is partitioned to a 2x2 rectangular space with 4 elements and 9 nodes. The nodal space is represented by a distgrid with 9 indices. A Field is created on locally owned nodes on each PET. Therefore, the created Field has 9 data points globally. The mesh object can be represented by the picture below. For more information on Mesh creation, please see Section **??**.

```
            Mesh Ids

2.0   7 ------- 8 -------- 9
      |         |          |
      |    3    |    4     |
      |         |          |
1.0   4 ------- 5 -------- 6
      |         |          |
      |    1    |    2     |
      |         |          |
0.0   1 ------- 2 -------- 3

     0.0       1.0        2.0

    Node Ids at corners
    Element Ids in centers


           Mesh Owners

2.0   2 ------- 2 -------- 3
      |         |          |
      |    2    |    3     |
      |         |          |
1.0   0 ------- 0 -------- 1
      |         |          |
      |    0    |    1     |
      |         |          |
0.0   0 ------- 0 -------- 1

     0.0       1.0        2.0

    Node Owners at corners
    Element Owners in centers
```

```
  ! Create Mesh structure in 1 step
  mesh=ESMF_MeshCreate(parametricDim=2,spatialDim=2, &
         nodeIds=nodeIds, nodeCoords=nodeCoords, &
```

```
        nodeOwners=nodeOwners, elementIds=elemIds,&
        elementTypes=elemTypes, elementConn=elemConn, &
        rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! Field is created on the 1 dimensional nodal distgrid. On
! each PET, Field is created on the locally owned nodes.
field = ESMF_FieldCreate(mesh, typekind=ESMF_TYPEKIND_I4, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

26.3.19 Create a Field from a Mesh and arrayspec

In this example, an `ESMF_Field` is created from an `ESMF_Mesh` and an `ESMF_Arrayspec`. The mesh object is on a Euclidean surface that is partitioned to a 2x2 rectangular space with 4 elements and 9 nodes. The nodal space is represented by a distgrid with 9 indices. Field is created on locally owned nodes on each PET. Therefore, the created Field has 9 data points globally. The mesh object can be represented by the picture below. For more information on Mesh creation, please see Section **??**.

```
! Create Mesh structure in 1 step
mesh=ESMF_MeshCreate(parametricDim=2,spatialDim=2, &
        nodeIds=nodeIds, nodeCoords=nodeCoords, &
        nodeOwners=nodeOwners, elementIds=elemIds,&
        elementTypes=elemTypes, elementConn=elemConn, &
        rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

call ESMF_ArraySpecSet(arrayspec, 1, ESMF_TYPEKIND_I4, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! Field is created on the 1 dimensional nodal distgrid. On
! each PET, Field is created on the locally owned nodes.
field = ESMF_FieldCreate(mesh, arrayspec, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

26.3.20 Create a Field from a Mesh and an Array

In this example, an `ESMF_Field` is created from an `ESMF_Mesh` and an `ESMF_Array`. The mesh object is created in the previous example and the array object is retrieved from the field created in the previous example too.

```
call ESMF_MeshGet(mesh, nodalDistgrid=distgrid, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
array = ESMF_ArrayCreate(distgrid=distgrid, arrayspec=arrayspec, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
! query the array from the previous example
call ESMF_FieldGet(field, array=array, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
! create a Field from a mesh and an array
field1 = ESMF_FieldCreate(mesh, array, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

26.3.21 Create a Field from a Mesh and an ArraySpec with optional features

In this example, an ESMF_Field is created from an ESMF_Mesh and an ESMF_ArraySpec. The mesh object is created in the previous example. The Field is also created with optional arguments such as ungridded dimensions and dimension mapping.

In this example, the mesh is mapped to the 2nd dimension of the ESMF_Field, with its first dimension being the ungridded dimension with bounds 1,3.

```
call ESMF_ArraySpecSet(arrayspec, 2, ESMF_TYPEKIND_I4, rc=rc)
field = ESMF_FieldCreate(mesh, arrayspec=arrayspec, gridToFieldMap=(/2/), &
    ungriddedLBound=(/1/), ungriddedUBound=(/3/), rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

26.3.22 Create a Field with replicated dimensions

In this example an ESMF_Field with replicated dimension is created from an ESMF_Grid and an ESMF_Arrayspec. A user can also use other ESMF_FieldCreate() methods to create replicated dimension Field, this example illustrates the key concepts and use of a replicated dimension Field.

Normally gridToFieldMap argument in ESMF_FieldCreate() should not contain 0 value entries. However, for a Field with replicated dimension, a 0 entry in gridToFieldMap indicates the corresponding Grid dimension is replicated in the Field. In such a Field, the rank of the Field is no longer necessarily greater than its Grid rank. An example will make this clear. We will start by creating Distgrid and Grid.

```
! create 4D distgrid
distgrid = ESMF_DistGridCreate(minIndex=(/1,1,1,1/), &
    maxIndex=(/6,4,6,4/), regDecomp=(/2,1,2,1/), rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! create 4D grid on top of the 4D distgrid
grid = ESMF_GridCreate(distgrid=distgrid, name="grid", rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! create 3D arrayspec
call ESMF_ArraySpecSet(arrayspec, 3, ESMF_TYPEKIND_R8, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

In this example, a user creates a 3D Field with replicated dimension replicated along the 2nd and 4th dimension of its underlying 4D Grid. In addition, the 2nd dimension of the Field is ungridded (why?). The 1st and 3rd dimensions of the Field have halos.

```
! create field, 2nd and 4th dimensions of the Grid are replicated
field = ESMF_FieldCreate(grid, arrayspec, indexflag=ESMF_INDEX_DELOCAL, &
    gridToFieldMap=(/1,0,2,0/), &
    ungriddedLBound=(/1/), ungriddedUBound=(/4/), &
    totalLWidth=(/1,1/), totalUWidth=(/4,5/), &
    staggerloc=ESMF_STAGGERLOC_CORNER, &
    rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

```
! get basic information from the field
call ESMF_FieldGet(field, grid=grid1, array=array, typekind=typekind, &
    dimCount=dimCount, staggerloc=lstaggerloc, &
    gridToFieldMap=lgridToFieldMap, ungriddedLBound=lungriddedLBound, &
    ungriddedUBound=lungriddedUBound, totalLWidth=ltotalLWidth, &
    totalUWidth=ltotalUWidth, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! get bounds information from the field
call ESMF_FieldGet(field, localDe=0, farrayPtr=farray, &
    exclusiveLBound=felb, exclusiveUBound=feub, exclusiveCount=fec, &
    computationalLBound=fclb, computationalUBound=fcub, &
    computationalCount=fcc, totalLBound=ftlb, totalUBound=ftub, &
    totalCount=ftc, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

Next we verify that the field and array bounds agree with each other

```
call ESMF_ArrayGet(array, rank=arank, dimCount=adimCount, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

gridrank_repdim = 0
do i = 1, size(gridToFieldMap)
    if(gridToFieldMap(i) == 0) gridrank_repdim = gridrank_repdim + 1
enddo
```

Number of undistributed dimension of the array *X* is computed from total rank of the array *A*, the dimension count of its underlying distgrid *B* and number of replicated dimension in the distgrid *C*. We have the following formula: X = A - (B - C)

```
allocate(audlb(arank-adimCount+gridrank_repdim), &
    audub(arank-adimCount+gridrank_repdim))
call ESMF_ArrayGet(array, exclusiveLBound=aelb, exclusiveUBound=aeub, &
    computationalLBound=aclb, computationalUBound=acub, &
    totalLBound=atlb, totalUBound=atub, &
    undistLBound=audlb, undistUBound=audub, &
    rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! verify the ungridded bounds from field match
! undistributed bounds from its underlying array
do i = 1, arank-adimCount
    if(lungriddedLBound(i) .ne. audlb(i) ) &
        rc = ESMF_FAILURE
enddo
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

do i = 1, arank-adimCount
    if(lungriddedUBound(i) .ne. audub(i) ) &
        rc = ESMF_FAILURE
```

```
enddo
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

We then verify the data in the replicated dimension Field can be updated and accessed.

```
do ik = ftlb(3), ftub(3)
 do ij = ftlb(2), ftub(2)
  do ii = ftlb(1), ftub(1)
    farray(ii,ij,ik) = ii+ij*2+ik
  enddo
 enddo
enddo
! access and verify
call ESMF_FieldGet(field, localDe=0, farrayPtr=farray1, &
    rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
do ik = ftlb(3), ftub(3)
 do ij = ftlb(2), ftub(2)
  do ii = ftlb(1), ftub(1)
    n = ii+ij*2+ik
    if(farray1(ii,ij,ik) .ne. n ) rc = ESMF_FAILURE
  enddo
 enddo
enddo
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! release resources
call ESMF_FieldDestroy(field)
call ESMF_GridDestroy(grid)
call ESMF_DistGridDestroy(distgrid)
```

26.3.23 Create a Field on an arbitrarily distributed Grid

With the introduction of Field on arbitrarily distributed Grid, Field has two kinds of dimension count: one associated geometrical (or physical) dimensionality, the other one associated with its memory index space representation. Field and Grid dimCount reflect the physical index space of the objects. A new type of dimCount rank should be added to both of these entities. The rank gives the number of dimensions of the memory index space of the objects. This would be the dimension of the pointer pulled out of Field and the size of the bounds vector, for example.

For non-arbitrary Grids rank=dimCount, but for grids and fields with arbitrary dimensions rank = dimCount - (number of Arb dims) + 1 (Internally Field can use the Arb info from the grid to create the mapping from the Field Array to the DistGrid)

When creating a Field size(GridToFieldMap)=dimCount for both Arb and Non-arb grids This array specifies the mapping of Field to Grid identically for both Arb and Nonarb grids If a zero occurs in an entry corresponding to any arbitrary dimension, then a zero must occur in every entry corresponding to an arbitrary dimension (i.e. all arbitrary dimensions must either be all replicated or all not replicated, they can't be broken apart).

In this example an `ESMF_Field` is created from an arbitrarily distributed `ESMF_Grid` and an `ESMF_Arrayspec`. A user can also use other `ESMF_FieldCreate()` methods to create such a Field, this example illustrates the key concepts and use of Field on arbitrary distributed Grid.

The Grid is 3 dimensional in physics index space but the first two dimension are collapsed into a single memory index space. Thus the resulting Field is 3D in physics index space and 2D in memory index space. This is made obvious with the 2D arrayspec used to create this Field.

```
! create a 3D grid with the first 2 dimensions collapsed
! and arbitrarily distributed
grid3d = ESMF_GridCreateNoPeriDim(coordTypeKind=ESMF_TYPEKIND_R8, &
  minIndex=(/1,1,1/), maxIndex=(/xdim, ydim,zdim/), &
  arbIndexList=localArbIndex,arbIndexCount=localArbIndexCount, &
  name="arb3dgrid", rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! create a 2D arrayspec
call ESMF_ArraySpecSet(arrayspec2D, rank=2, typekind=ESMF_TYPEKIND_R4, &
     rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! create a 2D Field using the Grid and the arrayspec
field = ESMF_FieldCreate(grid3d, arrayspec2D, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

call ESMF_FieldGet(field, rank=rank, dimCount=dimCount, &
                   rc=rc)
if (myPet .eq. 0) print *, 'Field rank, dimCount', &
                           rank, dimCount
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! verify that the dimension counts are correct
if (rank .ne. 2) correct = .false.
if (dimCount .ne. 3) correct = .false.
```

26.3.24 Create a Field on an arbitrarily distributed Grid with replicated dimensions & ungridded bounds

The next example is slightly more complicated in that the Field also contains one ungridded dimension and its gridded dimension is replicated on the arbitrarily distributed dimension of the Grid.

The same 3D Grid and 2D arrayspec in the previous example are used but a gridToFieldMap argument is supplied to the `ESMF_FieldCreate()` call. The first 2 entries of the map are 0, the last (3rd) entry is 1. The 3rd dimension of the Grid is mapped to the first dimension of the Field, this dimension is then replicated on the arbitrarily distributed dimensions of the Grid. In addition, the Field also has one ungridded dimension. Thus the final dimension count of the Field is 2 in both physics and memory index space.

```
field = ESMF_FieldCreate(grid3d, arrayspec2D,gridToFieldMap=(/0,0,1/), &
        ungriddedLBound=(/1/), ungriddedUBound=(/10/),rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

call ESMF_FieldGet(field, rank=rank, dimCount=dimCount, &
                   rc=rc)
if (myPet .eq. 0) print *, 'Field rank, dimCount', &
                           rank, dimCount
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

if (rank .ne. 2) correct = .false.
```

```
if (dimCount .ne. 2) correct = .false.
```

26.3.25 Field regridding

This section describes the Field regrid methods. For an in depth description of ESMF regridding and the options available please see Section 24.2.

The basic flow of ESMF Field regridding is as follows. First a source and destination geometry object are created, depending on the regrid method they can be either a Grid, a Mesh, an XGrid, or a LocStream. Next Fields are built on the source and destination grid objects. These Fields are then passed into `ESMF_FieldRegridStore()`. The user can either get a sparse matrix from this call and/or a `routeHandle`. If the user gets the sparse matrix then they are responsible for deallocating it, but other than that can use it as they wish. The `routeHandle` can be used in the `ESMF_FieldRegrid()` call to perform the actual interpolation of data from the source to the destination field. This interpolation can be repeated for the same set of Fields as long as the coordinates at the staggerloc involved in the regridding in the associated grid object don't change. The same `routeHandle` can also be used between any pair of Fields that matches the original pari in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability. However, if you want the routehandle to be the same interpolation between the grid objects upon which the Fields are built as was calculated with the original `ESMF_FieldRegridStore()` call, then there are additional constraints on the grid objects. To be the same interpolation, the grid objects upon which the Fields are build must contain the same coordinates at the stagger locations involved in the regridding as the original source and destination Fields used in the `ESMF_FieldRegridStore()` call. The routehandle represents the interpolation between the grid objects as they were during the `ESMF_FieldRegridStore()` call. So if the coordinates at the stagger location in the grid objects change, a new call to `ESMF_FieldRegridStore()` is necessary to compute the interpolation between that new set of coordinates. When finished with the `routeHandle` `ESMF_FieldRegridRelease()` should be used to free the associated memory.

The following example demonstrates doing a regrid operation between two Fields.

```
! (Create source Grid, Mesh, XGrid, or LocStream.)
! (Create srcField on the above.)

! (Create destination Grid, Mesh, XGrid, or LocStream.)
! (Create dstField on the above.)

! Create the routeHandle which encodes the communication and
! information necessary for the regrid sparse matrix multiply.
call ESMF_FieldRegridStore(srcField=srcField, dstField=dstField, &
                           routeHandle=routeHandle, rc=localrc)



! Can loop here regridding from srcField to dstField
! do i=1,....

     ! (Put data into srcField)

     ! Use the routeHandle to regrid data from srcField to dstField.
     ! As described above, the same routeHandle can be used to
     ! regrid a large class of different source and destination Fields.
     call ESMF_FieldRegrid(srcField, dstField, routeHandle, rc=localrc)
```

```
!    (Use data in dstField)

! enddo


! Free the buffers and data associated with the routeHandle.
call ESMF_FieldRegridRelease(routeHandle, rc=localrc)
```

26.3.26 Field regrid with masking

As before, to create the sparse matrix regrid operator we call the `ESMF_FieldRegridStore()` routine. However, in this case we apply masking to the regrid operation. The mask value for each index location in the Grids may be set using the `ESMF_GridAddItem()` call (see Section 31.3.17 and Section 31.3.18). Mask values may be set independently for the source and destination Grids. If no mask values have been set in a Grid, then it is assumed no masking should be used for that Grid. The `srcMaskValues` parameter allows the user to set the list of values which indicate that a source location should be masked out. The `dstMaskValues` parameter allows the user to set the list of values which indicate that a destination location should be masked out. The absence of one of these parameters indicates that no masking should be used for that Field (e.g no `srcMaskValue` parameter indicates that source masking shouldn't occur). The `unmappedaction` flag may be used with or without masking and indicates what should occur if destination points can not be mapped to a source cell. Here the `ESMF_UNMAPPEDACTION_IGNORE` value indicates that unmapped destination points are to be ignored and no sparse matrix entries should be generated for them.

```
call ESMF_FieldRegridStore(srcField=srcField, srcMaskValues=(/1/),        &
                           dstField=dstField, dstMaskValues=(/1/),        &
                           unmappedaction=ESMF_UNMAPPEDACTION_IGNORE, &
                           routeHandle=routeHandle,                       &
                           regridmethod=ESMF_REGRIDMETHOD_BILINEAR,      &
                           rc=localrc)
```

The `ESMF_FieldRegrid` and `ESMF_FieldRegridRelease` calls may then be applied as in the previous example.

26.3.27 Field regrid example: Mesh to Mesh

This example demonstrates the regridding process between Fields created on Meshes. First the Meshes are created. This example omits the setup of the arrays describing the Mesh, but please see Section **??** for examples of this. After creation Fields are constructed on the Meshes, and then ESMF_FieldRegridStore() is called to construct a RouteHandle implementing the regrid operation. Finally, ESMF_FieldRegrid() is called with the Fields and the RouteHandle to do the interpolation between the source Field and destination Field. Note the coordinates of the source and destination Mesh should be in degrees.

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Create Source Mesh
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
! Create the Mesh structure.
! For brevity's sake, the code to fill the Mesh creation
! arrays is omitted from this example. However, here
! is a brief description of the arrays:
! srcNodeIds    - the global ids for the src nodes
! srcNodeCoords - the coordinates for the src nodes
! srcNodeOwners - which PET owns each src node
! srcElemIds    - the global ids of the src elements
! srcElemTypes  - the topological shape of each src element
! srcElemConn   - how to connect the nodes to form the elements
!                 in the source mesh
! Several examples of setting up these arrays can be seen in
! the Mesh Section "Mesh Creation".
srcMesh=ESMF_MeshCreate(parametricDim=2,spatialDim=2, &
       nodeIds=srcNodeIds, nodeCoords=srcNodeCoords, &
       nodeOwners=srcNodeOwners, elementIds=srcElemIds,&
       elementTypes=srcElemTypes, elementConn=srcElemConn, rc=rc)

if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)



!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Create and Fill Source Field
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Set description of source Field
call ESMF_ArraySpecSet(arrayspec, 1, ESMF_TYPEKIND_R8, rc=rc)

if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! Create source Field
srcField = ESMF_FieldCreate(srcMesh, arrayspec, &
                      name="source", rc=rc)

if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! Get source Field data pointer to put data into
call ESMF_FieldGet(srcField, 0, fptr1D,  rc=rc)

if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! Get number of local nodes to allocate space
! to hold local node coordinates
call ESMF_MeshGet(srcMesh, &
       numOwnedNodes=numOwnedNodes, rc=rc)

if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! Allocate space to hold local node coordinates
! (spatial dimension of Mesh*number of local nodes)
allocate(ownedNodeCoords(2*numOwnedNodes))

! Get local node coordinates
call ESMF_MeshGet(srcMesh, &
```

```
       ownedNodeCoords=ownedNodeCoords, rc=rc)

if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! Set the source Field to the function 20.0+x+y
do i=1,numOwnedNodes
  ! Get coordinates
  x=ownedNodeCoords(2*i-1)
  y=ownedNodeCoords(2*i)

 ! Set source function
 fptr1D(i) = 20.0+x+y
enddo

! Deallocate local node coordinates
deallocate(ownedNodeCoords)


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Create Destination Mesh
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Create the Mesh structure.
! For brevity's sake, the code to fill the Mesh creation
! arrays is omitted from this example. However, here
! is a brief description of the arrays:
! dstNodeIds    - the global ids for the dst nodes
! dstNodeCoords - the coordinates for the dst nodes
! dstNodeOwners - which PET owns each dst node
! dstElemIds    - the global ids of the dst elements
! dstElemTypes  - the topological shape of each dst element
! dstElemConn   - how to connect the nodes to form the elements
!                 in the destination mesh
! Several examples of setting up these arrays can be seen in
! the Mesh Section "Mesh Creation".
dstMesh=ESMF_MeshCreate(parametricDim=2,spatialDim=2, &
       nodeIds=dstNodeIds, nodeCoords=dstNodeCoords, &
       nodeOwners=dstNodeOwners, elementIds=dstElemIds,&
       elementTypes=dstElemTypes, elementConn=dstElemConn, rc=rc)

if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Create Destination Field
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Set description of source Field
call ESMF_ArraySpecSet(arrayspec, 1, ESMF_TYPEKIND_R8, rc=rc)

if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! Create destination Field
dstField = ESMF_FieldCreate(dstMesh, arrayspec, &
                     name="destination", rc=rc)
```

```
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Do Regrid
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Compute RouteHandle which contains the regrid operation
call ESMF_FieldRegridStore( &
        srcField, &
        dstField=dstField, &
        routeHandle=routeHandle, &
        regridmethod=ESMF_REGRIDMETHOD_BILINEAR, &
        rc=rc)

if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! Perform Regrid operation moving data from srcField to dstField
call ESMF_FieldRegrid(srcField, dstField, routeHandle, rc=rc)


if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! dstField now contains the interpolated data.
! If the Meshes don't change, then routeHandle
! may be used repeatedly to interpolate from
! srcField to dstField.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!


! User code to use the routeHandle, Fields, and
! Meshes goes here before they are freed below.


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Free the objects created in the example.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Free the RouteHandle
call ESMF_FieldRegridRelease(routeHandle, rc=rc)

if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! Free the Fields
call ESMF_FieldDestroy(srcField, rc=rc)

if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

call ESMF_FieldDestroy(dstField, rc=rc)

if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! Free the Meshes
call ESMF_MeshDestroy(dstMesh, rc=rc)
```

```
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

call ESMF_MeshDestroy(srcMesh, rc=rc)

if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

26.3.28 Gather Field data onto root PET

User can use `ESMF_FieldGather` interface to gather Field data from multiple PETs onto a single root PET. This interface is overloaded by type, kind, and rank.

Note that the implementation of Scatter and Gather is not sequence index based. If the Field is built on arbitrarily distributed Grid, Mesh, LocStream or XGrid, Gather will not gather data to rootPet from source data points corresponding to the sequence index on the rootPet. Instead Gather will gather a contiguous memory range from source PET to rootPet. The size of the memory range is equal to the number of data elements on the source PET. Vice versa for the Scatter operation. In this case, the user should use `ESMF_FieldRedist` to achieve the same data operation result. For examples how to use `ESMF_FieldRedist` to perform Gather and Scatter, please refer to 26.3.32 and 26.3.31.

In this example, we first create a 2D Field, then use `ESMF_FieldGather` to collect all the data in this Field into a data pointer on PET 0.

```
! Get current VM and pet number
call ESMF_VMGetCurrent(vm, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

call ESMF_VMGet(vm, localPet=lpe, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! Create a 2D Grid and use this grid to create a Field
! farray is the Fortran data array that contains data on each PET.
grid = ESMF_GridCreateNoPeriDim(minIndex=(/1,1/), maxIndex=(/10,20/), &
    regDecomp=(/2,2/), &
    name="grid", rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

field = ESMF_FieldCreate(grid, typekind=ESMF_TYPEKIND_I4, rc=localrc)
if (localrc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)


call ESMF_FieldGet(field, farrayPtr=fptr, rc=localrc)
if (localrc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
!---------Initialize pet specific field data---------------
!    1        5         10
! 1  +--------+---------+
!    |        |         |
!    |   0    |    1    |
!    |        |         |
! 10 +--------+---------+
!    |        |         |
!    |   2    |    3    |
```

```
!    |        |         |
! 20 +--------+---------+
fptr = lpe

! allocate the Fortran data array on PET 0 to store gathered data
if(lpe .eq. 0) then
  allocate (farrayDst(10,20))
else
  allocate (farrayDst(0,0))
end if
call ESMF_FieldGather(field, farrayDst, rootPet=0, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! check that the values gathered on rootPet are correct
if(lpe .eq. 0) then
   do i = 1, 5
      do j = 1, 10
         if(farrayDst(i, j) .ne. 0) localrc=ESMF_FAILURE
      enddo
   enddo
  if (localrc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
   do i = 6, 10
      do j = 1, 10
         if(farrayDst(i, j) .ne. 1) localrc=ESMF_FAILURE
      enddo
   enddo
  if (localrc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
   do i = 1, 5
      do j = 11, 20
         if(farrayDst(i, j) .ne. 2) localrc=ESMF_FAILURE
      enddo
   enddo
  if (localrc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
   do i = 6, 10
      do j = 11, 20
         if(farrayDst(i, j) .ne. 3) localrc=ESMF_FAILURE
      enddo
   enddo
  if (localrc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif

! destroy all objects created in this example to prevent memory leak
call ESMF_FieldDestroy(field, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
call ESMF_GridDestroy(grid, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
if(lpe .eq. 0) deallocate(farrayDst)
```

26.3.29 Scatter Field data from root PET onto its set of joint PETs

User can use `ESMF_FieldScatter` interface to scatter Field data from root PET onto its set of joint PETs. This interface is overloaded by type, kind, and rank.

In this example, we first create a 2D Field, then use `ESMF_FieldScatter` to scatter the data from a data array

located on PET 0 onto this Field.

```
! Create a 2D Grid and use this grid to create a Field
! farray is the Fortran data array that contains data on each PET.
grid = ESMF_GridCreateNoPeriDim(minIndex=(/1,1/), maxIndex=(/10,20/), &
    regDecomp=(/2,2/), &
    name="grid", rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

field = ESMF_FieldCreate(grid, typekind=ESMF_TYPEKIND_I4, rc=localrc)
if (localrc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! initialize values to be scattered
!    1        5         10
! 1  +--------+---------+
!    |        |         |
!    |   0    |    1    |
!    |        |         |
! 10 +--------+---------+
!    |        |         |
!    |   2    |    3    |
!    |        |         |
! 20 +--------+---------+
if(lpe .eq. 0) then
    allocate(farraySrc(10,20))
    farraySrc(1:5,1:10) = 0
    farraySrc(6:10,1:10) = 1
    farraySrc(1:5,11:20) = 2
    farraySrc(6:10,11:20) = 3
else
  allocate (farraySrc(0,0))
endif

! scatter the data onto individual PETs of the Field
call ESMF_FieldScatter(field, farraySrc, rootPet=0, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

call ESMF_FieldGet(field, localDe=0, farrayPtr=fptr, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! verify that the scattered data is properly distributed
do i = lbound(fptr, 1), ubound(fptr, 1)
    do j = lbound(fptr, 2), ubound(fptr, 2)
        if(fptr(i, j) .ne. lpe) localrc = ESMF_FAILURE
    enddo
    if (localrc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
enddo

! destroy all objects created in this example to prevent memory leak
call ESMF_FieldDestroy(field, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
call ESMF_GridDestroy(grid, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
if(lpe .eq. 0) deallocate(farraySrc)
```

26.3.30 Redistribute data from source Field to destination Field

User can use ESMF_FieldRedist interface to redistribute data from source Field to destination Field. This interface is overloaded by type and kind; In the version of ESMF_FieldRedist without factor argument, a default value of 1 is used.

In this example, we first create two 1D Fields, a source Field and a destination Field. Then we use ESMF_FieldRedist to redistribute data from source Field to destination Field.

```
! Get current VM and pet number
call ESMF_VMGetCurrent(vm, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

call ESMF_VMGet(vm, localPet=localPet, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! create grid
distgrid = ESMF_DistGridCreate(minIndex=(/1/), maxIndex=(/16/), &
        regDecomp=(/4/), &
        rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

grid = ESMF_GridCreate(distgrid=distgrid, &
    name="grid", rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! create srcField
! +--------+--------+--------+--------+
!      0        1        2        3            ! value
! 1        4        8        12       16       ! bounds
srcField = ESMF_FieldCreate(grid, typekind=ESMF_TYPEKIND_I4, &
  indexflag=ESMF_INDEX_DELOCAL, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

call ESMF_FieldGet(srcField, farrayPtr=srcfptr, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

srcfptr(:) = localPet

! create dstField
! +--------+--------+--------+--------+
!      0        0        0        0            ! value
! 1        4        8        12       16       ! bounds
dstField = ESMF_FieldCreate(grid, typekind=ESMF_TYPEKIND_I4, &
  indexflag=ESMF_INDEX_DELOCAL, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

call ESMF_FieldGet(dstField, farrayPtr=dstfptr, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

dstfptr(:) = 0

! perform redist
! 1. setup routehandle from source Field to destination Field
call ESMF_FieldRedistStore(srcField, dstField, routehandle, rc=rc)
```

```
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! 2. use precomputed routehandle to redistribute data
call ESMF_FieldRedist(srcfield, dstField, routehandle, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! verify redist
call ESMF_FieldGet(dstField, localDe=0, farrayPtr=fptr, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! Verify that the redistributed data in dstField is correct.
! Before the redist op, the dst Field contains all 0.
! The redist op reset the values to the PE value, verify this is the case.
do i = lbound(fptr, 1), ubound(fptr, 1)
    if(fptr(i) .ne. localPet) localrc = ESMF_FAILURE
enddo
if (localrc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

Field redistribution can also be performed between different Field pairs that match the original Fields in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_I4, rank=2, rc=rc)
```

Create two fields with ungridded dimensions using the Grid created previously. The new Field pair has matching number of elements. The ungridded dimension is mapped to the first dimension of either Field.

```
srcFieldA = ESMF_FieldCreate(grid, arrayspec, gridToFieldMap=(/2/), &
    ungriddedLBound=(/1/), ungriddedUBound=(/10/), rc=rc)


dstFieldA = ESMF_FieldCreate(grid, arrayspec, gridToFieldMap=(/2/), &
    ungriddedLBound=(/1/), ungriddedUBound=(/10/), rc=rc)
```

Using the previously computed routehandle, the Fields can be redistributed.

```
call ESMF_FieldRedist(srcfieldA, dstFieldA, routehandle, rc=rc)


call ESMF_FieldRedistRelease(routehandle, rc=rc)
```

26.3.31 FieldRedist as a form of scatter involving arbitrary distribution

User can use `ESMF_FieldRedist` interface to redistribute data from source Field to destination Field, where the destination Field is built on an arbitrarily distributed structure, e.g. `ESMF_Mesh`. The underlying mechanism is explained in section 28.2.19.

In this example, we will create 2 one dimensional Fields, the src Field has a regular decomposition and holds all its data on a single PET, in this case PET 0. The destination Field is built on a Mesh which is itself built on an arbitrarily distributed distgrid. Then we use `ESMF_FieldRedist` to redistribute data from source Field to destination Field, similar to a traditional scatter operation.

The src Field only has data on PET 0 where it is sequentially initialized, i.e. 1,2,3...This data will be redistributed (or scattered) from PET 0 to the destination Field arbitrarily distributed on all the PETs.

```
! a one dimensional grid whose elements are all located on PET 0
distgrid = ESMF_DistGridCreate(minIndex=(/1/), maxIndex=(/9/), &
    regDecomp=(/1/), &
    rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
grid = ESMF_GridCreate(distgrid=distgrid, &
    indexflag=ESMF_INDEX_DELOCAL, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

srcField = ESMF_FieldCreate(grid, typekind=ESMF_TYPEKIND_I4, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! initialize the source data
if (localPet == 0) then
    call ESMF_FieldGet(srcField, farrayPtr=srcfptr, rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
    do i = 1, 9
        srcfptr(i) = i
    enddo
endif
```

For more information on Mesh creation, user can refer to Mesh examples section or Field creation on Mesh example for more details.

```
  ! Create Mesh structure
  mesh=ESMF_MeshCreate(parametricDim=2,spatialDim=2, &
         nodeIds=nodeIds, nodeCoords=nodeCoords, &
         nodeOwners=nodeOwners, elementIds=elemIds,&
         elementTypes=elemTypes, elementConn=elemConn, &
         rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

Create the destination Field on the Mesh that is arbitrarily distributed on all the PETs.

```
  dstField = ESMF_FieldCreate(mesh, typekind=ESMF_TYPEKIND_I4, rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

Perform the redistribution from source Field to destination Field.

```
 call ESMF_FieldRedistStore(srcField, dstField, &
         routehandle=routehandle, rc=rc)
 if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

```
call ESMF_FieldRedist(srcField, dstField, routehandle=routehandle, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

We can now verify that the sequentially initialized source data is scattered on to the destination Field. The data has been scattered onto the destination Field with the following distribution.

```
4 elements on PET 0:  1 2 4 5
2 elements on PET 1:  3 6
2 elements on PET 2:  7 8
1 element  on PET 3:  9
```

Because the redistribution is index based, the elements also corresponds to the index space of Mesh in the destination Field.

```
call ESMF_FieldGet(dstField, farrayPtr=dstfptr, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

The scatter operation is successful. Since the routehandle computed with `ESMF_FieldRedistStore` can be reused, user can use the same routehandle to scatter multiple source Fields from a single PET to multiple destination Fields distributed on all PETs. The `gathering` operation is just the opposite of the demonstrated `scattering` operation, where a user would redist from a source Field distributed on multiple PETs to a destination Field that only has data storage on a single PET.

Now it's time to release all the resources.

```
call ESMF_FieldRedistRelease(routehandle=routehandle, rc=rc)
```

26.3.32 FieldRedist as a form of gather involving arbitrary distribution

Similarly, one can use the same approach to gather the data from an arbitrary distribution to a non-arbitrary distribution. This concept is demonstrated by using the previous Fields but the data operation is reversed. This time data is gathered from the Field built on the mesh to the Field that has only data allocation on rootPet.

First a FieldRedist routehandle is created from the Field built on Mesh to the Field that has only data allocation on rootPet.

```
call ESMF_FieldRedistStore(dstField, srcField, routehandle=routehandle, &
     rc=rc)
```

Perform FieldRedist, this will gather the data points from the Field built on mesh to the data pointer on the rootPet (default to 0) stored in the srcField.

```
call ESMF_FieldRedist(dstField, srcField, routehandle=routehandle, rc=rc)
```

Release the routehandle used for the gather operation.

```
call ESMF_FieldRedistRelease(routehandle=routehandle, rc=rc)
```

26.3.33 Sparse matrix multiplication from source Field to destination Field

The `ESMF_FieldSMM()` interface can be used to perform sparse matrix multiplication from source Field to destination Field. This interface is overloaded by type and kind;

In this example, we first create two 1D Fields, a source Field and a destination Field. Then we use `ESMF_FieldSMM` to perform sparse matrix multiplication from source Field to destination Field.

The source and destination Field data are arranged such that each of the 4 PETs has 4 data elements. Moreover, the source Field has all its data elements initialized to a linear function based on local PET number. Then collectively on each PET, a SMM according to the following formula is preformed:
$dstField(i) = i * srcField(i), i = 1...4$

Because source Field data are initialized to a linear function based on local PET number, the formula predicts that the result destination Field data on each PET is 1,2,3,4. This is verified in the example.

Section 28.2.18 provides a detailed discussion of the sparse matrix multiplication operation implemented in ESMF.

```
! Get current VM and pet number
call ESMF_VMGetCurrent(vm, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

call ESMF_VMGet(vm, localPet=lpe, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! create distgrid and grid
distgrid = ESMF_DistGridCreate(minIndex=(/1/), maxIndex=(/16/), &
    regDecomp=(/4/), &
    rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

grid = ESMF_GridCreate(distgrid=distgrid, &
    name="grid", rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

call ESMF_GridGetFieldBounds(grid, localDe=0, totalCount=fa_shape, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! create src\_farray, srcArray, and srcField
! +--------+--------+--------+--------+
!      1        2        3        4            ! value
! 1        4        8        12       16       ! bounds
allocate(src_farray(fa_shape(1)) )
src_farray = lpe+1
srcArray = ESMF_ArrayCreate(distgrid, src_farray, &
            indexflag=ESMF_INDEX_DELOCAL, rc=rc)
```

```
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

srcField = ESMF_FieldCreate(grid, srcArray, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! create dst_farray, dstArray, and dstField
! +--------+--------+--------+--------+
!      0        0        0        0           ! value
! 1        4        8        12       16      ! bounds
allocate(dst_farray(fa_shape(1)) )
dst_farray = 0
dstArray = ESMF_ArrayCreate(distgrid, dst_farray, &
           indexflag=ESMF_INDEX_DELOCAL, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

dstField = ESMF_FieldCreate(grid, dstArray, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! perform sparse matrix multiplication
! 1. setup routehandle from source Field to destination Field
! initialize factorList and factorIndexList
allocate(factorList(4))
allocate(factorIndexList(2,4))
factorList = (/1,2,3,4/)
factorIndexList(1,:) = (/lpe*4+1,lpe*4+2,lpe*4+3,lpe*4+4/)
factorIndexList(2,:) = (/lpe*4+1,lpe*4+2,lpe*4+3,lpe*4+4/)

call ESMF_FieldSMMStore(srcField, dstField, routehandle, &
    factorList, factorIndexList, rc=localrc)
if (localrc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! 2. use precomputed routehandle to perform SMM
call ESMF_FieldSMM(srcfield, dstField, routehandle, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! verify sparse matrix multiplication
call ESMF_FieldGet(dstField, localDe=0, farrayPtr=fptr, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

! Verify that the result data in dstField is correct.
! Before the SMM op, the dst Field contains all 0.
! The SMM op reset the values to the index value, verify this is the case.
! +--------+--------+--------+--------+
!  1 2 3 4  2 4 6 8  3 6 9 12  4 8 12 16      ! value
! 1        4        8        12       16      ! bounds
do i = lbound(fptr, 1), ubound(fptr, 1)
    if(fptr(i) /= i*(lpe+1)) rc = ESMF_FAILURE
enddo
```

Field sparse matrix multiplication can also be applied between Fields that matche the original Fields in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_I4, rank=2, rc=rc)
```

Create two fields with ungridded dimensions using the Grid created previously. The new Field pair has matching number of elements. The ungridded dimension is mapped to the first dimension of either Field.

```
srcFieldA = ESMF_FieldCreate(grid, arrayspec, gridToFieldMap=(/2/), &
    ungriddedLBound=(/1/), ungriddedUBound=(/10/), rc=rc)


dstFieldA = ESMF_FieldCreate(grid, arrayspec, gridToFieldMap=(/2/), &
    ungriddedLBound=(/1/), ungriddedUBound=(/10/), rc=rc)
```

Using the previously computed routehandle, the sparse matrix multiplication can be performed between the Fields.

```
call ESMF_FieldSMM(srcfieldA, dstFieldA, routehandle, rc=rc)


! release route handle
call ESMF_FieldSMMRelease(routehandle, rc=rc)
```

In the following discussion, we demonstrate how to set up a SMM routehandle between a pair of Fields that are different in number of gridded dimensions and the size of those gridded dimensions. The source Field has a 1D decomposition with 16 total elements; the destination Field has a 2D decomposition with 12 total elements. For ease of understanding of the actual matrix calculation, a global indexing scheme is used.

```
distgrid = ESMF_DistGridCreate(minIndex=(/1/), maxIndex=(/16/), &
    indexflag=ESMF_INDEX_GLOBAL, &
    regDecomp=(/4/), &
    rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

grid = ESMF_GridCreate(distgrid=distgrid, &
    indexflag=ESMF_INDEX_GLOBAL, &
    name="grid", rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

call ESMF_GridGetFieldBounds(grid, localDe=0, totalLBound=tlb, &
                  totalUBound=tub, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

create 1D src_farray, srcArray, and srcField

```
+  PET0  +  PET1  +  PET2  +  PET3  +
+--------+--------+--------+--------+
     1        2        3        4            ! value
1        4        8        12       16       ! bounds of seq indices
```

```
allocate(src_farray2(tlb(1):tub(1)) )
src_farray2 = lpe+1
srcArray = ESMF_ArrayCreate(distgrid, src_farray2, &
             indexflag=ESMF_INDEX_GLOBAL, &
  rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
!print *, lpe, '+', tlb, tub, '+', src_farray2

srcField = ESMF_FieldCreate(grid, srcArray, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

Create 2D dstField on the following distribution (numbers are the sequence indices):

```
+  PET0  +  PET1  +  PET2  +  PET3  +
+--------+--------+--------+--------+
|        |        |        |        |
|   1    |   4    |   7    |   10   |
|        |        |        |        |
+--------+--------+--------+--------+
|        |        |        |        |
|   2    |   5    |   8    |   11   |
|        |        |        |        |
+--------+--------+--------+--------+
|        |        |        |        |
|   3    |   6    |   9    |   12   |
|        |        |        |        |
+--------+--------+--------+--------+
```

```
! Create the destination Grid
dstGrid = ESMF_GridCreateNoPeriDim(minIndex=(/1,1/), maxIndex=(/3,4/), &
  indexflag = ESMF_INDEX_GLOBAL, &
  regDecomp = (/1,4/), &
  rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

dstField = ESMF_FieldCreate(dstGrid, typekind=ESMF_TYPEKIND_R4, &
  indexflag=ESMF_INDEX_GLOBAL, &
  rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

Perform sparse matrix multiplication dst_i = $M_{i,j}$ * src_j First setup routehandle from source Field to destination Field with prescribed factorList and factorIndexList.

The sparse matrix is of size 12x16, however only the following entries are filled:

```
M(3,1) = 0.1
M(3,10) = 0.4
M(8,2) = 0.25
M(8,16) = 0.5
```

```
M(12,1) = 0.3
M(12,16) = 0.7
```

By the definition of matrix calculation, the 8th element on PET2 in the dstField equals to 0.25*srcField(2) + 0.5*srcField(16) = 0.25*1+0.5*4=2.25 For simplicity, we will load the factorList and factorIndexList on PET 0 and 1, the SMMStore engine will load balance the parameters on all 4 PETs internally for optimal performance.

```
if(lpe == 0) then
  allocate(factorList(3), factorIndexList(2,3))
  factorList=(/0.1,0.4,0.25/)
  factorIndexList(1,:)=(/1,10,2/)
  factorIndexList(2,:)=(/3,3,8/)
  call ESMF_FieldSMMStore(srcField, dstField, routehandle=routehandle, &
      factorList=factorList, factorIndexList=factorIndexList, rc=localrc)
  if (localrc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
else if(lpe == 1) then
  allocate(factorList(3), factorIndexList(2,3))
  factorList=(/0.5,0.3,0.7/)
  factorIndexList(1,:)=(/16,1,16/)
  factorIndexList(2,:)=(/8,12,12/)
  call ESMF_FieldSMMStore(srcField, dstField, routehandle=routehandle, &
      factorList=factorList, factorIndexList=factorIndexList, rc=localrc)
  if (localrc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
else
  call ESMF_FieldSMMStore(srcField, dstField, routehandle=routehandle, &
      rc=localrc)
  if (localrc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif

! 2. use precomputed routehandle to perform SMM
call ESMF_FieldSMM(srcfield, dstField, routehandle=routehandle, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

26.3.34 Field Halo solving a domain decomposed heat transfer problem

The `ESMF_FieldHalo()` interface can be used to perform halo updates for a Field. This eases communication programming from a user perspective. By definition, the user program only needs to update locally owned exclusive region in each domain, then call FieldHalo to communicate the values in the halo region from/to neighboring domain elements. In this example, we solve a 1D heat transfer problem: $u_t = \alpha^2 u_{xx}$ with the initial condition $u(0, x) = 20$ and boundary conditions $u(t, 0) = 10, u(t, 1) = 40$. The temperature field u is represented by a `ESMF_Field`. A finite difference explicit time stepping scheme is employed. During each time step, FieldHalo update is called to communicate values in the halo region to neighboring domain elements. The steady state (as $t \to \infty$) solution is a linear temperature profile along x. The numerical solution is an approximation of the steady state solution. It can be verified to represent a linear temperature profile.

Section 28.2.15 provides a discussion of the halo operation implemented in `ESMF_Array`.

```
! create 1D distgrid and grid decomposed according to the following diagram:
! +------------+   +----------------+   +---------------+   +--------------+
! |   DE 0   |  |   |  |   DE 1    |  |   |  |   DE 2   |  |   |  |   DE 3    |
! |  1 x 16 |  |   |  |  1 x 16  |  |   |  |  1 x 16 |  |   |  |  1 x 16   |
```

```
! |         | 1|<->|1 |          | 1|<->|1 |         | 1|<->|1 |           |
! |         |  |   |  |          |  |   |  |         |  |   |  |           |
! +---------+   +----------------+   +---------------+   +--------------+
    distgrid = ESMF_DistGridCreate(minIndex=(/1/), maxIndex=(/npx/), &
        regDecomp=(/4/), rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

    grid = ESMF_GridCreate(distgrid=distgrid, name="grid", rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

    ! set up initial condition and boundary conditions of the
    ! temperature Field
    if(lpe == 0) then
        allocate(fptr(17), tmp_farray(17))
        fptr = 20.
        fptr(1) = 10.
        tmp_farray(1) = 10.
        startx = 2
        endx = 16

        field = ESMF_FieldCreate(grid, fptr, totalUWidth=(/1/), &
                name="temperature", rc=rc)
        if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
    else if(lpe == 3) then
        allocate(fptr(17), tmp_farray(17))
        fptr = 20.
        fptr(17) = 40.
        tmp_farray(17) = 40.
        startx = 2
        endx = 16

        field = ESMF_FieldCreate(grid, fptr, totalLWidth=(/1/), &
                name="temperature", rc=rc)
        if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
    else
        allocate(fptr(18), tmp_farray(18))
        fptr = 20.
        startx = 2
        endx = 17

        field = ESMF_FieldCreate(grid, fptr, &
            totalLWidth=(/1/), totalUWidth=(/1/), name="temperature", rc=rc)
        if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
    endif

    ! compute the halo update routehandle of the decomposed temperature Field
    call ESMF_FieldHaloStore(field, routehandle=routehandle, rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

    dt = 0.01
    dx = 1./npx
    alpha = 0.1

    ! Employ explicit time stepping
    ! Solution converges after about 9000 steps based on apriori knowledge.
```

```
! The result is a linear temperature profile stored in field.
do iter = 1, 9000
 ! only elements in the exclusive region are updated locally
 ! in each domain
 do i = startx, endx
   tmp_farray(i) = &
   fptr(i)+alpha*alpha*dt/dx/dx*(fptr(i+1)-2.*fptr(i)+fptr(i-1))
  enddo
  fptr = tmp_farray
 ! call halo update to communicate the values in the halo region to
 ! neighboring domains
 call ESMF_FieldHalo(field, routehandle=routehandle, rc=rc)
 if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
enddo

! release the halo routehandle
call ESMF_FieldHaloRelease(routehandle, rc=rc)
```

26.4 Restrictions and Future Work

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

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

3. **No vector Fields.** ESMF does not currently support storage of multiple vector Field components in the same Field component, although that support is planned. At this time users need to create a separate Field object to represent each vector component.

26.5 Design and Implementation Notes

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

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

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

In this way the overhead of communication is avoided, at the cost of more difficulty in diagnosing program bugs which result from not all PETs executing the same create calls.

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

26.6 Class API

26.6.1 ESMF_FieldAssignment(=) - Field assignment

INTERFACE:

```
interface assignment(=)
field1 = field2
```

ARGUMENTS:

```
type(ESMF_Field) :: field1
type(ESMF_Field) :: field2
```

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:

Assign field1 as an alias to the same ESMF Field object in memory as field2. If field2 is invalid, then field1 will be equally invalid after the assignment.

The arguments are:

field1 The `ESMF_Field` object on the left hand side of the assignment.

field2 The `ESMF_Field` object on the right hand side of the assignment.

26.6.2 ESMF_FieldOperator(==) - Field equality operator

INTERFACE:

```
interface operator(==)
if (field1 == field2) then ... endif
OR
result = (field1 == field2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field1
type(ESMF_Field), intent(in) :: field2
```

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:

Test whether field1 and field2 are valid aliases to the same ESMF Field object in memory. For a more general comparison of two ESMF Fields, going beyond the simple alias test, the ESMF_FieldMatch() function (not yet implemented) must be used.

The arguments are:

field1 The `ESMF_Field` object on the left hand side of the equality operation.

field2 The `ESMF_Field` object on the right hand side of the equality operation.

26.6.3 ESMF_FieldOperator(/=) - Field not equal operator

INTERFACE:

```
interface operator(/=)
if (field1 /= field2) then ... endif
OR
result = (field1 /= field2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field1
type(ESMF_Field), intent(in) :: field2
```

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:

Test whether field1 and field2 are *not* valid aliases to the same ESMF Field object in memory. For a more general comparison of two ESMF Fields, going beyond the simple alias test, the ESMF_FieldMatch() function (not yet implemented) must be used.

The arguments are:

field1 The `ESMF_Field` object on the left hand side of the non-equality operation.

field2 The `ESMF_Field` object on the right hand side of the non-equality operation.

26.6.4 ESMF_FieldCopy - Copy data from one Field object to another

INTERFACE:

```
   subroutine ESMF_FieldCopy(fieldOut, fieldIn, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Copy data from one `ESMF_Field` object to another.

The arguments are:

fieldOut `ESMF_Field` object into which to copy the data. The incoming `fieldOut` must already references a matching memory allocation.

fieldIn `ESMF_Field` object that holds the data to be copied.

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

26.6.5 ESMF_FieldCreate - Create a Field from Grid and typekind

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateGridTKR(grid, typekind, &
  indexflag, staggerloc, gridToFieldMap, ungriddedLBound, ungriddedUBound, &
  totalLWidth, totalUWidth, pinflag, name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateGridTKR
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
type(ESMF_TypeKind_Flag),intent(in) :: typekind
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_Index_Flag), intent(in), optional :: indexflag
type(ESMF_StaggerLoc), intent(in), optional :: staggerloc
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
integer, intent(in), optional :: totalLWidth(:)
integer, intent(in), optional :: totalUWidth(:)
type(ESMF_Pin_Flag), intent(in), optional :: pinflag
character (len=*), intent(in), optional :: name
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:

  8.1.0 Added argument `pinflag` to provide access to DE sharing between PETs.

DESCRIPTION:

Create an `ESMF_Field` and allocate space internally for an `ESMF_Array`. Return a new `ESMF_Field`. For an example and associated documentation using this method see section 26.3.4.

The arguments are:

grid `ESMF_Grid` object.

typekind The typekind of the Field. See section **??** for a list of valid typekind options.

[indexflag] Indicate how DE-local indices are defined. By default each DE's exclusive region is placed to start at the local index space origin, i.e. (1, 1, ..., 1). Alternatively the DE-local index space can be aligned with the global index space, if a global index space is well defined by the associated Grid. See section **??** for a list of valid indexflag options. The default indexflag value is the one stored in then `ESMF_Grid` object. Currently it is erroneous to specify an indexflag different from the one stored in the `ESMF_Grid` object. The default value is `ESMF_INDEX_DELOCAL`

[staggerloc] Stagger location of data in grid cells. For valid predefined values see section 31.2.6. To create a custom stagger location see section 31.3.25. The default value is `ESMF_STAGGERLOC_CENTER`.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `field` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `field` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should be max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[pinflag] Specify which type of resource DEs are pinned to. See section **??** for a list of valid pinning options. The default is to pin DEs to PETs, i.e. only the PET on which a DE was created considers the DE as local.

[name] Field name.

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

26.6.6 ESMF_FieldCreate - Create a Field from Grid and ArraySpec

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateGridArraySpec(grid, arrayspec, &
  indexflag, staggerloc, gridToFieldMap, ungriddedLBound, &
  ungriddedUBound, totalLWidth, totalUWidth, pinflag, name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateGridArraySpec
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
type(ESMF_ArraySpec), intent(in) :: arrayspec
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_Index_Flag), intent(in), optional :: indexflag
type(ESMF_StaggerLoc), intent(in), optional :: staggerloc
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
integer, intent(in), optional :: totalLWidth(:)
integer, intent(in), optional :: totalUWidth(:)
type(ESMF_Pin_Flag), intent(in), optional :: pinflag
character (len=*), intent(in), optional :: name
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:

  8.1.0 Added argument `pinflag` to provide access to DE sharing between PETs.

DESCRIPTION:

Create an `ESMF_Field` and allocate space internally for an `ESMF_Array`. Return a new `ESMF_Field`. For an example and associated documentation using this method see section 26.3.5.

The arguments are:

grid `ESMF_Grid` object.

arrayspec Data type and kind specification.

[indexflag] Indicate how DE-local indices are defined. By default each DE's exclusive region is placed to start at the local index space origin, i.e. (1, 1, ..., 1). Alternatively the DE-local index space can be aligned with the global index space, if a global index space is well defined by the associated Grid. See section **??** for a list of valid indexflag options. The default indexflag value is the one stored in then `ESMF_Grid` object. Currently it is erroneous to specify an indexflag different from the one stored in the `ESMF_Grid` object.The default value is `ESMF_INDEX_DELOCAL`

[staggerloc] Stagger location of data in grid cells. For valid predefined values see section 31.2.6. To create a custom stagger location see section 31.3.25. The default value is ESMF_STAGGERLOC_CENTER.

[gridToFieldMap] List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the field by specifying the appropriate field dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the field in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the field rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total field dimensions less the dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should be max(totalLWidth + totalUWidth + computationalCount, exclusiveCount).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should max(totalLWidth + totalUWidth + computationalCount, exclusiveCount).

[pinflag] Specify which type of resource DEs are pinned to. See section **??** for a list of valid pinning options. The default is to pin DEs to PETs, i.e. only the PET on which a DE was created considers the DE as local.

[name] Field name.

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

26.6.7 ESMF_FieldCreate - Create a Field from Grid and Array

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateGridArray(grid, array, datacopyflag, &
  staggerloc, gridToFieldMap, ungriddedLBound, ungriddedUBound, &
  totalLWidth, totalUWidth, name, vm, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateGridArray
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
type(ESMF_Array), intent(in) :: array
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
type(ESMF_StaggerLoc), intent(in), optional :: staggerloc
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
integer, intent(in), optional :: totalLWidth(:)
integer, intent(in), optional :: totalUWidth(:)
character (len = *), intent(in), optional :: name
type(ESMF_VM), intent(in), 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:

  8.0.0 Added argument `vm` to support object creation on a different VM than that of the current context.

DESCRIPTION:

Create an `ESMF_Field`. This version of creation assumes the data exists already and is being passed in through an `ESMF_Array`. For an example and associated documentation using this method see section 26.3.6.

The arguments are:

grid `ESMF_Grid` object.

array `ESMF_Array` object.

[datacopyflag] Indicates whether to copy the contents of the `array` or reference it directly. For valid values see **??**. The default is `ESMF_DATACOPY_REFERENCE`.

[staggerloc] Stagger location of data in grid cells. For valid predefined values see section 31.2.6. To create a custom stagger location see section 31.3.25. The default value is `ESMF_STAGGERLOC_CENTER`.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `field` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `field` rank. It is erroneous to specify the same `gridToFieldMap`

entry multiple times. The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should be max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[name] Field name.

[vm] If present, the Field object is constructed on the specified `ESMF_VM` object. The default is to construct on the VM of the current component context.

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

26.6.8 ESMF_FieldCreate - Create a Field from Grid and Fortran array

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateGridData<rank><type><kind>(grid, &
farray, indexflag, datacopyflag, staggerloc, &
gridToFieldMap, ungriddedLBound, ungriddedUBound, &
totalLWidth, totalUWidth, name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateGridData<rank><type><kind>
```

ARGUMENTS:

```
   type(ESMF_Grid), intent(in) :: grid
   <type> (ESMF_KIND_<kind>),intent(in), target :: farray(<rank>)
   type(ESMF_Index_Flag), intent(in) :: indexflag
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
   type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
   type(ESMF_StaggerLoc), intent(in), optional :: staggerloc
   integer, intent(in), optional :: gridToFieldMap(:)
   integer, intent(in), optional :: ungriddedLBound(:)
   integer, intent(in), optional :: ungriddedUBound(:)
   integer, intent(in), optional :: totalLWidth(:)
   integer, intent(in), optional :: totalUWidth(:)
   character (len=*), intent(in), optional :: name
   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:

Create an `ESMF_Field` from a Fortran data array and `ESMF_Grid`. The Fortran data pointer inside `ESMF_Field` can be queried but deallocating the retrieved data pointer is not allowed. For examples and associated documentation regarding this method see section 26.3.11, 26.3.13, 26.3.14, 26.3.15, and 26.3.9.

The arguments are:

grid `ESMF_Grid` object.

farray Native Fortran data array to be copied/referenced in the Field The Field dimension (dimCount) will be the same as the dimCount for the `farray`.

indexflag Indicate how DE-local indices are defined. See section **??** for a list of valid indexflag options. Currently it is erroneous to specify an indexflag different from the one stored in the `ESMF_Grid` object.

[datacopyflag] Whether to copy the contents of the `farray` or reference it directly. For valid values see **??**. The default is `ESMF_DATACOPY_REFERENCE`.

[staggerloc] Stagger location of data in grid cells. For valid predefined values see section 31.2.6. To create a custom stagger location see section 31.3.25. The default value is `ESMF_STAGGERLOC_CENTER`.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `farray` by specifying the appropriate `farray` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `farray` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `farray` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `farray` dimensions less the total (distributed + undistributed) dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the farray. Permutations of the order of dimensions are handled via individual communication methods. For example, an undistributed dimension can be remapped to a distributed dimension as part of the `ESMF_ArrayRedist()` operation. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `farray`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `farray`.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the `farray`. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the `farray`. That is, for each gridded dimension the `farray` size should be max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the `farray`. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the `farray`. That is, for each gridded dimension the `farray` size should max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[name] Field name.

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

26.6.9 ESMF_FieldCreate - Create a Field from Grid and Fortran array pointer

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateGridDataPtr<rank><type><kind>(grid, &
farrayPtr, datacopyflag, staggerloc, gridToFieldMap, &
totalLWidth, totalUWidth, name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateGridDataPtr<rank><type><kind>
```

ARGUMENTS:

```
type(ESMF_Grid), intent(in) :: grid
<type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
type(ESMF_StaggerLoc), intent(in), optional :: staggerloc
integer, intent(in), optional :: gridToFieldMap(:)
```

```
integer, intent(in), optional :: totalLWidth(:)
integer, intent(in), optional :: totalUWidth(:)
character (len=*), intent(in), optional :: name
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:

Create an ESMF_Field from a Fortran data pointer and ESMF_Grid. The Fortran data pointer inside ESMF_Field can be queried and deallocated when datacopyflag is ESMF_DATACOPY_REFERENCE. Note that the ESMF_FieldDestroy call does not deallocate the Fortran data pointer in this case. This gives user more flexibility over memory management.

For examples and associated documentation regarding this method see section 26.3.12, 26.3.13, 26.3.14, 26.3.15, and 26.3.9.

The arguments are:

grid ESMF_Grid object.

farrayPtr Native Fortran data pointer to be copied/referenced in the Field The Field dimension (dimCount) will be the same as the dimCount for the farrayPtr.

[datacopyflag] Whether to copy the contents of the farrayPtr or reference it directly. For valid values see **??**. The default is ESMF_DATACOPY_REFERENCE.

[staggerloc] Stagger location of data in grid cells. For valid predefined values see section 31.2.6. To create a custom stagger location see section 31.3.25. The default value is ESMF_STAGGERLOC_CENTER.

[gridToFieldMap] List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the farrayPtr by specifying the appropriate farrayPtr dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the farrayPtr in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the farrayPtr rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total farrayPtr dimensions less the total (distributed + undistributed) dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the farrayPtr. Permutations of the order of dimensions are handled via individual communication methods. For example, an undistributed dimension can be remapped to a distributed dimension as part of the ESMF_ArrayRedist() operation. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the farrayPtr. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the farrayPtr. That is, for each gridded dimension the farrayPtr size should be max(totalLWidth + totalUWidth + computationalCount, exclusiveCount).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the farrayPtr. Values

default to 0. If values for totalUWidth are specified they must be reflected in the size of the farrayPtr. That is, for each gridded dimension the farrayPtr size should max(totalLWidth + totalUWidth + computationalCount, exclusiveCount).

[name] Field name.

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

26.6.10 ESMF_FieldCreate - Create a Field from LocStream and typekind

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateLSTKR(locstream, typekind, &
  gridToFieldMap, ungriddedLBound, ungriddedUBound, pinflag, name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateLSTKR
```

ARGUMENTS:

```
type(ESMF_LocStream), intent(in) :: locstream
type(ESMF_TypeKind_Flag),intent(in) :: typekind
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
type(ESMF_Pin_Flag), intent(in), optional :: pinflag
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an ESMF_Field and allocate space internally for an ESMF_Array. Return a new ESMF_Field. For an example and associated documentation using this method see section 26.3.16.

The arguments are:

locstream ESMF_LocStream object.

typekind The typekind of the Field. See section **??** for a list of valid typekind options.

[gridToFieldMap] List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the field by specifying the appropriate field dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the field in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the field rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total field dimensions less the dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the field. If the

Field dimCount is less than the LocStream dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular LocStream dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[pinflag] Specify which type of resource DEs are pinned to. See section **??** for a list of valid pinning options. The default is to pin DEs to PETs, i.e. only the PET on which a DE was created considers the DE as local.

[name] Field name.

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

26.6.11 ESMF_FieldCreate - Create a Field from LocStream and ArraySpec

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateLSArraySpec(locstream, arrayspec, &
  gridToFieldMap, ungriddedLBound, ungriddedUBound, pinflag, name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateLSArraySpec
```

ARGUMENTS:

```
type(ESMF_LocStream), intent(in) :: locstream
type(ESMF_ArraySpec), intent(in) :: arrayspec
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
type(ESMF_Pin_Flag), intent(in), optional :: pinflag
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an `ESMF_Field` and allocate space internally for an `ESMF_Array`. Return a new `ESMF_Field`. For an example and associated documentation using this method see section 26.3.17.

The arguments are:

locstream ESMF_LocStream object.

arrayspec Data type and kind specification.

[gridToFieldMap] List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the field by specifying the appropriate field dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the field in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the field rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total field dimensions less the dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the LocStream dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular LocStream dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[pinflag] Specify which type of resource DEs are pinned to. See section **??** for a list of valid pinning options. The default is to pin DEs to PETs, i.e. only the PET on which a DE was created considers the DE as local.

[name] Field name.

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

26.6.12 ESMF_FieldCreate - Create a Field from LocStream and Array

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateLSArray(locstream, array, &
  datacopyflag, gridToFieldMap, ungriddedLBound, ungriddedUBound, &
  name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateLSArray
```

ARGUMENTS:

```
type(ESMF_LocStream), intent(in) :: locstream
type(ESMF_Array), intent(in) :: array
```

```
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
    integer, intent(in), optional :: gridToFieldMap(:)
    integer, intent(in), optional :: ungriddedLBound(:)
    integer, intent(in), optional :: ungriddedUBound(:)
    character (len = *), intent(in), optional :: name
    integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an ESMF_Field. This version of creation assumes the data exists already and is being passed in through an ESMF_Array. For an example and associated documentation using this method see section 26.3.6.

The arguments are:

locstream ESMF_LocStream object.

array ESMF_Array object.

[datacopyflag] Indicates whether to copy the contents of the array or reference it directly. For valid values see **??**. The default is ESMF_DATACOPY_REFERENCE.

[gridToFieldMap] List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the field by specifying the appropriate field dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the field in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the field rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total field dimensions less the dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the LocStream dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular LocStream dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[name] Field name.

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

26.6.13 ESMF_FieldCreate - Create a Field from LocStream and Fortran array

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateLSData<rank><type><kind>(locstream, farray, &
indexflag, datacopyflag, gridToFieldMap, ungriddedLBound, &
ungriddedUBound, name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateLSData<rank><type><kind>
```

ARGUMENTS:

```
type(ESMF_LocStream), intent(in) :: locstream
<type> (ESMF_KIND_<kind>),intent(in), target :: farray(<rank>)
type(ESMF_Index_Flag), intent(in) :: indexflag
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an `ESMF_Field` from a Fortran data array and `ESMF_LocStream`. The Fortran data pointer inside `ESMF_Field` can be queried but deallocating the retrieved data pointer is not allowed.

The arguments are:

locstream `ESMF_LocStream` object.

farray Native Fortran data array to be copied/referenced in the Field The Field dimension (dimCount) will be the same as the dimCount for the `farray`.

indexflag Indicate how DE-local indices are defined. See section **??** for a list of valid indexflag options.

[datacopyflag] Whether to copy the contents of the `farray` or reference directly. For valid values see **??**. The default is `ESMF_DATACOPY_REFERENCE`.

[gridToFieldMap] List with number of elements equal to the `locstream`'s dimCount. The list elements map each dimension of the `locstream` to a dimension in the `farray` by specifying the appropriate `farray` dimension index. The default is to map all of the `locstream`'s dimensions against the lowest dimensions of the `farray` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `farray` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `farray` dimensions less the total (distributed + undistributed) dimensions in the `locstream`. Unlocstreamded dimensions must be in the same order they are stored in the farray. Permutations of the order of dimensions are handled via individual communication methods. For example, an undistributed dimension can be remapped to a distributed dimension as part of the `ESMF_ArrayRedist()` operation. If the Field dimCount is less than the LocStream dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular LocStream dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than locstream dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `farray`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than locstream dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `farray`.

[name] Field name.

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

26.6.14 ESMF_FieldCreate - Create a Field from LocStream and Fortran array pointer

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateLSDataPtr<rank><type><kind>(locstream, &
farrayPtr, datacopyflag, gridToFieldMap, &
name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateLSDataPtr<rank><type><kind>
```

ARGUMENTS:

```
  type(ESMF_LocStream), intent(in) :: locstream
  <type> (ESMF_KIND_<kind>),pointer :: farrayPtr(<rank>)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
  integer, intent(in), optional :: gridToFieldMap(:)
  character (len=*), intent(in), optional :: name
  integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an `ESMF_Field` from a Fortran data pointer and `ESMF_LocStream`. The Fortran data pointer inside `ESMF_Field` can be queried and deallocated when datacopyflag is `ESMF_DATACOPY_REFERENCE`. Note that the `ESMF_FieldDestroy` call does not deallocate the Fortran data pointer in this case. This gives user more flexibility over memory management.

The arguments are:

locstream ESMF_LocStream object.

farrayPtr Native Fortran data pointer to be copied/referenced in the Field The Field dimension (dimCount) will be the same as the dimCount for the farrayPtr.

[datacopyflag] Whether to copy the contents of the farrayPtr or reference it directly. For valid values see **??**. The default is ESMF_DATACOPY_REFERENCE.

[gridToFieldMap] List with number of elements equal to the locstream's dimCount. The list elements map each dimension of the locstream to a dimension in the farrayPtr by specifying the appropriate farrayPtr dimension index. The default is to map all of the locstream's dimensions against the lowest dimensions of the farrayPtr in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the farrayPtr rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total farrayPtr dimensions less the total (distributed + undistributed) dimensions in the locstream. Unlocstreamded dimensions must be in the same order they are stored in the farrayPtr. Permutations of the order of dimensions are handled via individual communication methods. For example, an undistributed dimension can be remapped to a distributed dimension as part of the ESMF_ArrayRedist() operation. If the Field dimCount is less than the LocStream dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular LocStream dimension will be replicating the Field across the DEs along this direction.

[name] Field name.

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

26.6.15 ESMF_FieldCreate - Create a Field from Mesh and typekind

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateMeshTKR(mesh, typekind, indexflag, &
  meshloc, gridToFieldMap, ungriddedLBound, ungriddedUBound, &
  pinflag, name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateMeshTKR
```

ARGUMENTS:

```
type(ESMF_Mesh), intent(in) :: mesh
type(ESMF_TypeKind_Flag), intent(in) :: typekind
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_Index_Flag), intent(in), optional :: indexflag
type(ESMF_MeshLoc), intent(in), optional :: meshloc
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
type(ESMF_Pin_Flag), intent(in), optional :: pinflag
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an ESMF_Field and allocate space internally for an ESMF_Array. Return a new ESMF_Field. For an example and associated documentation using this method see section 26.3.18.

The arguments are:

mesh ESMF_Mesh object.

typekind The typekind of the Field. See section **??** for a list of valid typekind options.

[indexflag] Indicate how DE-local indices are defined. See section **??** for a list of valid indexflag options.

[meshloc] The part of the Mesh on which to build the Field. For valid predefined values see Section **??**. If not set, defaults to ESMF_MESHLOC_NODE.

[gridToFieldMap] List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the field by specifying the appropriate field dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the field in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the field rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total field dimensions less the dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the Mesh dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Mesh dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[pinflag] Specify which type of resource DEs are pinned to. See section **??** for a list of valid pinning options. The default is to pin DEs to PETs, i.e. only the PET on which a DE was created considers the DE as local.

[name] Field name.

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

26.6.16 ESMF_FieldCreate - Create a Field from Mesh and ArraySpec

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateMeshArraySpec(mesh, arrayspec, &
  indexflag, meshloc, gridToFieldMap, ungriddedLBound, ungriddedUBound, &
  pinflag, name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateMeshArraySpec
```

ARGUMENTS:

```
type(ESMF_Mesh), intent(in) :: mesh
type(ESMF_ArraySpec), intent(in) :: arrayspec
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_Index_Flag),intent(in), optional :: indexflag
type(ESMF_MeshLoc), intent(in), optional :: meshloc
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
type(ESMF_Pin_Flag), intent(in), optional :: pinflag
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an `ESMF_Field` and allocate space internally for an `ESMF_Array`. Return a new `ESMF_Field`. For an example and associated documentation using this method see section 26.3.19 and 26.3.21.

The arguments are:

mesh `ESMF_Mesh` object.

arrayspec Data type and kind specification.

[indexflag] Indicate how DE-local indices are defined. See section **??** for a list of valid indexflag options.

[meshloc] The part of the Mesh on which to build the Field. For valid predefined values see Section **??**. If not set, defaults to `ESMF_MESHLOC_NODE`.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `field` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `field` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the Mesh dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular Mesh dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[pinflag] Specify which type of resource DEs are pinned to. See section **??** for a list of valid pinning options. The default is to pin DEs to PETs, i.e. only the PET on which a DE was created considers the DE as local.

[name] Field name.

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

26.6.17 ESMF_FieldCreate - Create a Field from Mesh and Array

INTERFACE:

```
   ! Private name; call using ESMF_FieldCreate()
   function ESMF_FieldCreateMeshArray(mesh, array, &
     datacopyflag, meshloc, &
     gridToFieldMap, ungriddedLBound, ungriddedUBound, &
     name, vm, rc)
```

RETURN VALUE:

```
     type(ESMF_Field) :: ESMF_FieldCreateMeshArray
```

ARGUMENTS:

```
     type(ESMF_Mesh), intent(in) :: mesh
     type(ESMF_Array), intent(in) :: array
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
     type(ESMF_MeshLoc), intent(in), optional :: meshloc
     integer, intent(in), optional :: gridToFieldMap(:)
     integer, intent(in), optional :: ungriddedLBound(:)
     integer, intent(in), optional :: ungriddedUBound(:)
     character (len = *), intent(in), optional :: name
     type(ESMF_VM), intent(in), optional :: vm
     integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an `ESMF_Field`. This version of creation assumes the data exists already and is being passed in through an `ESMF_Array`. For an example and associated documentation using this method see section 26.3.20.

The arguments are:

mesh `ESMF_Mesh` object.

array `ESMF_Array` object.

[datacopyflag] Indicates whether to copy the contents of the `array` or reference it directly. For valid values see **??**. The default is `ESMF_DATACOPY_REFERENCE`.

[meshloc] The part of the Mesh on which to build the Field. For valid predefined values see Section **??**. If not set, defaults to `ESMF_MESHLOC_NODE`.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `field` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `field` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the Mesh dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular Mesh dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[name] Field name.

[vm] If present, the Field object is constructed on the specified `ESMF_VM` object. The default is to construct on the VM of the current component context.

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

26.6.18 ESMF_FieldCreate - Create a Field from Mesh and Fortran array

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateMeshData<rank><type><kind>(mesh, &
farray, indexflag, datacopyflag, meshloc, &
gridToFieldMap, ungriddedLBound, ungriddedUBound, name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateMeshData<rank><type><kind>
```

ARGUMENTS:

```
type(ESMF_Mesh), intent(in) :: mesh
<type> (ESMF_KIND_<kind>),intent(in), target :: farray(<rank>)
type(ESMF_Index_Flag), intent(in) :: indexflag
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
```

```
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
type(ESMF_MeshLoc), intent(in), optional :: meshloc
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an ESMF_Field from a Fortran data array and ESMF_Mesh. The Fortran data pointer inside ESMF_Field can be queried but deallocating the retrieved data pointer is not allowed.

The arguments are:

mesh ESMF_Mesh object.

farray Native Fortran data array to be copied/referenced in the Field The Field dimension (dimCount) will be the same as the dimCount for the farray.

indexflag Indicate how DE-local indices are defined. See section **??** for a list of valid indexflag options.

[datacopyflag] Whether to copy the contents of the farray or reference it directly. For valid values see **??**. The default is ESMF_DATACOPY_REFERENCE.

[meshloc] The part of the Mesh on which to build the Field. For valid predefined values see Section **??**. If not set, defaults to ESMF_MESHLOC_NODE.

[gridToFieldMap] List with number of elements equal to the mesh's dimCount. The list elements map each dimension of the mesh to a dimension in the farray by specifying the appropriate farray dimension index. The default is to map all of the mesh's dimensions against the lowest dimensions of the farray in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the farray rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total farray dimensions less the total (distributed + undistributed) dimensions in the mesh. Unmeshded dimensions must be in the same order they are stored in the farray. Permutations of the order of dimensions are handled via individual communication methods. For example, an undistributed dimension can be remapped to a distributed dimension as part of the ESMF_ArrayRedist() operation. If the Field dimCount is less than the Mesh dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Mesh dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than mesh dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the farray.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than mesh dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the farray.

[name] Field name.

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

26.6.19 ESMF_FieldCreate - Create a Field from Mesh and Fortran array pointer

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateMeshDataPtr<rank><type><kind>(mesh, &
farrayPtr, datacopyflag, meshloc, gridToFieldMap, &
name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateMeshDataPtr<rank><type><kind>
```

ARGUMENTS:

```
type(ESMF_Mesh), intent(in) :: mesh
<type> (ESMF_KIND_<kind>),pointer :: farrayPtr(<rank>)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
type(ESMF_MeshLoc), intent(in), optional :: meshloc
integer, intent(in), optional :: gridToFieldMap(:)
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an `ESMF_Field` from a Fortran data pointer and `ESMF_Mesh`. The Fortran data pointer inside `ESMF_Field` can be queried and deallocated when datacopyflag is `ESMF_DATACOPY_REFERENCE`. Note that the `ESMF_FieldDestroy` call does not deallocate the Fortran data pointer in this case. This gives user more flexibility over memory management.

The arguments are:

mesh `ESMF_Mesh` object.

farrayPtr Native Fortran data pointer to be copied/referenced in the Field The Field dimension (dimCount) will be the same as the dimCount for the `farrayPtr`.

[datacopyflag] Whether to copy the contents of the `farrayPtr` or reference it directly. For valid values see **??**. The default is `ESMF_DATACOPY_REFERENCE`.

[meshloc] The part of the Mesh on which to build the Field. For valid predefined values see Section **??**. If not set, defaults to `ESMF_MESHLOC_NODE`.

[gridToFieldMap] List with number of elements equal to the `mesh`'s dimCount. The list elements map each dimension of the `mesh` to a dimension in the `farrayPtr` by specifying the appropriate `farrayPtr` dimension index. The default is to map all of the `mesh`'s dimensions against the lowest dimensions of the `farrayPtr` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `farrayPtr` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total

farrayPtr dimensions less the total (distributed + undistributed) dimensions in the mesh. Unmeshded dimensions must be in the same order they are stored in the farrayPtr. Permutations of the order of dimensions are handled via individual communication methods. For example, an undistributed dimension can be remapped to a distributed dimension as part of the ESMF_ArrayRedist() operation. If the Field dimCount is less than the Mesh dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Mesh dimension will be replicating the Field across the DEs along this direction.

[name] Field name.

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

26.6.20 ESMF_FieldCreate - Create a Field from XGrid and typekind

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateXGTKR(xgrid, typekind, xgridside, &
  gridindex, gridToFieldMap, ungriddedLBound, ungriddedUBound, &
  pinflag, name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateXGTKR
```

ARGUMENTS:

```
type(ESMF_XGrid), intent(in) :: xgrid
type(ESMF_TypeKind_Flag), intent(in) :: typekind
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_XGridSide_Flag), intent(in), optional :: xgridside
integer, intent(in), optional :: gridindex
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
type(ESMF_Pin_Flag), intent(in), optional :: pinflag
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an ESMF_Field and allocate space internally for an ESMF_Array. Return a new ESMF_Field. For an example and associated documentation using this method see section 26.3.16.

The arguments are:

xgrid ESMF_XGrid object.

typekind The typekind of the Field. See section **??** for a list of valid typekind options.

[xgridside] Which side of the XGrid to create the Field on (either ESMF_XGRIDSIDE_A, ESMF_XGRIDSIDE_B, or ESMF_XGRIDSIDE_BALANCED). If not passed in then defaults to ESMF_XGRIDSIDE_BALANCED.

[gridindex] If xgridSide is ESMF_XGRIDSIDE_A or ESMF_XGRIDSIDE_B then this index tells which Grid on that side to create the Field on. If not provided, defaults to 1.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `field` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `field` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the XGrid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular XGrid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[pinflag] Specify which type of resource DEs are pinned to. See section **??** for a list of valid pinning options. The default is to pin DEs to PETs, i.e. only the PET on which a DE was created considers the DE as local.

[name] Field name.

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

26.6.21 ESMF_FieldCreate - Create a Field from XGrid and ArraySpec

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateXGArraySpec(xgrid, arrayspec, &
  xgridside, gridindex, gridToFieldMap, ungriddedLBound, ungriddedUBound, &
  pinflag, name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateXGArraySpec
```

ARGUMENTS:

```
    type(ESMF_XGrid), intent(in) :: xgrid
    type(ESMF_ArraySpec), intent(in) :: arrayspec
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_XGridSide_Flag), intent(in), optional :: xgridSide
    integer, intent(in), optional :: gridIndex
    integer, intent(in), optional :: gridToFieldMap(:)
    integer, intent(in), optional :: ungriddedLBound(:)
    integer, intent(in), optional :: ungriddedUBound(:)
    type(ESMF_Pin_Flag), intent(in), optional :: pinflag
    character (len=*), intent(in), optional :: name
    integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an ESMF_Field and allocate space internally for an ESMF_Array. Return a new ESMF_Field. For an example and associated documentation using this method see section 26.3.17.

The arguments are:

xgrid ESMF_XGrid object.

arrayspec Data type and kind specification.

[xgridside] Which side of the XGrid to create the Field on (either ESMF_XGRIDSIDE_A, ESMF_XGRIDSIDE_B, or ESMF_XGRIDSIDE_BALANCED). If not passed in then defaults to ESMF_XGRIDSIDE_BALANCED.

[gridindex] If xgridside is ESMF_XGRIDSIDE_A or ESMF_XGRIDSIDE_B then this index tells which Grid on that side to create the Field on. If not provided, defaults to 1.

[gridToFieldMap] List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the field by specifying the appropriate field dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the field in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the field rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total field dimensions less the dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the XGrid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular XGrid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[pinflag] Specify which type of resource DEs are pinned to. See section **??** for a list of valid pinning options. The default is to pin DEs to PETs, i.e. only the PET on which a DE was created considers the DE as local.

[name] Field name.

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

26.6.22 ESMF_FieldCreate - Create a Field from XGrid and Array

INTERFACE:

```
   ! Private name; call using ESMF_FieldCreate()
   function ESMF_FieldCreateXGArray(xgrid, array, &
     datacopyflag, xgridside, gridindex, &
     gridToFieldMap, ungriddedLBound, ungriddedUBound, &
     name, rc)
```

RETURN VALUE:

```
     type(ESMF_Field) :: ESMF_FieldCreateXGArray
```

ARGUMENTS:

```
     type(ESMF_XGrid), intent(in) :: xgrid
     type(ESMF_Array), intent(in) :: array
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
     type(ESMF_XGridSide_Flag), intent(in), optional :: xgridside
     integer, intent(in), optional :: gridindex
     integer, intent(in), optional :: gridToFieldMap(:)
     integer, intent(in), optional :: ungriddedLBound(:)
     integer, intent(in), optional :: ungriddedUBound(:)
     character (len = *), intent(in), optional :: name
     integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an `ESMF_Field`. This version of creation assumes the data exists already and is being passed in through an `ESMF_Array`. For an example and associated documentation using this method see section 26.3.6.

The arguments are:

xgrid `ESMF_XGrid` object.

array `ESMF_Array` object.

[datacopyflag] Indicates whether to copy the contents of the `array` or reference it directly. For valid values see **??**. The default is `ESMF_DATACOPY_REFERENCE`.

[xgridside] Which side of the XGrid to create the Field on (either ESMF_XGRIDSIDE_A, ESMF_XGRIDSIDE_B, or ESMF_XGRIDSIDE_BALANCED). If not passed in then defaults to ESMF_XGRIDSIDE_BALANCED.

[gridindex] If xgridSide is ESMF_XGRIDSIDE_A or ESMF_XGRIDSIDE_B then this index tells which Grid on that side to create the Field on. If not provided, defaults to 1.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `field` in sequence, i.e.

gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the field rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total field dimensions less the dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the XGrid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular XGrid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[name] Field name.

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

26.6.23 ESMF_FieldCreate - Create a Field from XGrid and Fortran array

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateXGData<rank><type><kind>(xgrid, &
farray, indexflag, datacopyflag, xgridside, gridindex, &
gridToFieldMap, ungriddedLBound, ungriddedUBound, name,&
rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateXGData<rank><type><kind>
```

ARGUMENTS:

```
type(ESMF_XGrid), intent(in) :: xgrid
<type> (ESMF_KIND_<kind>), intent(in), target :: farray(<rank>)
type(ESMF_Index_Flag), intent(in) :: indexflag
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
type(ESMF_XGridSide_Flag), intent(in), optional :: xgridside
integer, intent(in), optional :: gridindex
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
```

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

DESCRIPTION:

Create an ESMF_Field from a Fortran data array and ESMF_Xgrid. The Fortran data pointer inside ESMF_Field can be queried but deallocating the retrieved data pointer is not allowed.

The arguments are:

xgrid ESMF_XGrid object.

farray Native Fortran data array to be copied/referenced in the Field The Field dimension (dimCount) will be the same as the dimCount for the farray.

indexflag Indicate how DE-local indices are defined. See section **??** for a list of valid indexflag options.

[datacopyflag] Whether to copy the contents of the farray or reference directly. For valid values see **??**. The default is ESMF_DATACOPY_REFERENCE.

[xgridside] Which side of the XGrid to create the Field on (either ESMF_XGRIDSIDE_A, ESMF_XGRIDSIDE_B, or ESMF_XGRIDSIDE_BALANCED). If not passed in then defaults to ESMF_XGRIDSIDE_BALANCED.

[gridindex] If xgridside is ESMF_XGRIDSIDE_A or ESMF_XGRIDSIDE_B then this index tells which Grid on that side to create the Field on. If not provided, defaults to 1.

[gridToFieldMap] List with number of elements equal to the xgrid's dimCount. The list elements map each dimension of the xgrid to a dimension in the farray by specifying the appropriate farray dimension index. The default is to map all of the xgrid's dimensions against the lowest dimensions of the farray in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the farray rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total farray dimensions less the total (distributed + undistributed) dimensions in the xgrid. Unxgridded dimensions must be in the same order they are stored in the farray. Permutations of the order of dimensions are handled via individual communication methods. For example, an undistributed dimension can be remapped to a distributed dimension as part of the ESMF_ArrayRedist() operation. If the Field dimCount is less than the Xgrid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Xgrid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than xgrid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the farray.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than xgrid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the farray.

[name] Field name.

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

26.6.24 ESMF_FieldCreate - Create a Field from XGrid and Fortran array pointer

INTERFACE:

```
! Private name; call using ESMF_FieldCreate()
function ESMF_FieldCreateXGDataPtr<rank><type><kind>(xgrid, farrayPtr, &
datacopyflag, xgridside, &
gridindex, gridToFieldMap, name, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldCreateXGDataPtr<rank><type><kind>
```

ARGUMENTS:

```
type(ESMF_XGrid), intent(in) :: xgrid
<type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
type(ESMF_XGridSide_Flag), intent(in), optional :: xgridside
integer, intent(in), optional :: gridindex
integer, intent(in), optional :: gridToFieldMap(:)
character (len=*), intent(in), optional :: name
integer, intent(out), optional :: rc
```

DESCRIPTION:

Create an ESMF_Field from a Fortran data pointer and ESMF_Xgrid. The Fortran data pointer inside ESMF_Field can be queried and deallocated when datacopyflag is ESMF_DATACOPY_REFERENCE. Note that the ESMF_FieldDestroy call does not deallocate the Fortran data pointer in this case. This gives user more flexibility over memory management.

The arguments are:

xgrid ESMF_XGrid object.

farrayPtr Native Fortran data pointer to be copied/referenced in the Field The Field dimension (dimCount) will be the same as the dimCount for the farrayPtr.

[datacopyflag] Whether to copy the contents of the farrayPtr or reference it directly. For valid values see **??**. The default is ESMF_DATACOPY_REFERENCE.

[xgridside] Which side of the XGrid to create the Field on (either ESMF_XGRIDSIDE_A, ESMF_XGRIDSIDE_B, or ESMF_XGRIDSIDE_BALANCED). If not passed in then defaults to ESMF_XGRIDSIDE_BALANCED.

[gridindex] If xgridside is ESMF_XGRIDSIDE_A or ESMF_XGRIDSIDE_B then this index tells which Grid on that side to create the Field on. If not provided, defaults to 1.

[gridToFieldMap] List with number of elements equal to the xgrid's dimCount. The list elements map each dimension of the xgrid to a dimension in the farrayPtr by specifying the appropriate farrayPtr dimension index. The default is to map all of the xgrid's dimensions against the lowest dimensions of the farrayPtr

in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the farrayPtr rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total farrayPtr dimensions less the total (distributed + undistributed) dimensions in the xgrid. Unxgridded dimensions must be in the same order they are stored in the farrayPtr. Permutations of the order of dimensions are handled via individual communication methods. For example, an undistributed dimension can be remapped to a distributed dimension as part of the ESMF_ArrayRedist() operation. If the Field dimCount is less than the Xgrid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Xgrid dimension will be replicating the Field across the DEs along this direction.

[name] Field name.

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

26.6.25 ESMF_FieldDestroy - Release resources associated with a Field

INTERFACE:

```
subroutine ESMF_FieldDestroy(field, noGarbage, rc)
```

ARGUMENTS:

```
     type(ESMF_Field), intent(inout) :: field
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical, intent(in), optional :: noGarbage
     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 noGarbage. The argument provides a mechanism to override the default garbage collection mechanism when destroying an ESMF object.

DESCRIPTION:

Destroys the ESMF_Field, releasing the resources associated with the object.

If an ESMF_Grid is associated with field, it will not be released.

By default a small remnant of the object is kept in memory in order to prevent problems with dangling aliases. The default garbage collection mechanism can be overridden with the noGarbage argument.

The arguments are:

field ESMF_Field object.

[noGarbage] If set to .TRUE. the object will be fully destroyed and removed from the ESMF garbage collection system. Note however that under this condition ESMF cannot protect against accessing the destroyed object through dangling aliases – a situation which may lead to hard to debug application crashes.

It is generally recommended to leave the noGarbage argument set to .FALSE. (the default), and to take advantage of the ESMF garbage collection system which will prevent problems with dangling aliases or incorrect sequences of destroy calls. However this level of support requires that a small remnant of the object is kept in memory past the destroy call. This can lead to an unexpected increase in memory consumption over the course of execution in applications that use temporary ESMF objects. For situations where the repeated creation and destruction of temporary objects leads to memory issues, it is recommended to call with noGarbage set to .TRUE., fully removing the entire temporary object from memory.

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

26.6.26 ESMF_FieldEmptyComplete - Complete a Field from arrayspec

INTERFACE:

```
   ! Private name; call using ESMF_FieldEmptyComplete()
 subroutine ESMF_FieldEmptyCompAS(field, arrayspec, indexflag, &
  gridToFieldMap, ungriddedLBound, ungriddedUBound, totalLWidth, totalUWidth, &
  pinflag, rc)
```

ARGUMENTS:

```
  type(ESMF_Field), intent(inout) :: field
  type(ESMF_ArraySpec), intent(in) :: arrayspec
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_Index_Flag), intent(in), optional :: indexflag
  integer, intent(in), optional :: gridToFieldMap(:)
  integer, intent(in), optional :: ungriddedLBound(:)
  integer, intent(in), optional :: ungriddedUBound(:)
  integer, intent(in), optional :: totalLWidth(:)
  integer, intent(in), optional :: totalUWidth(:)
  type(ESMF_Pin_Flag), intent(in), optional :: pinflag
  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:

  8.1.0 Added argument pinflag to provide access to DE sharing between PETs.

DESCRIPTION:

Complete an ESMF_Field and allocate space internally for an ESMF_Array based on arrayspec. The input ESMF_Field must have a status of ESMF_FIELDSTATUS_GRIDSET. After this call the completed ESMF_Field has a status of ESMF_FIELDSTATUS_COMPLETE.

The arguments are:

field The input ESMF_Field with a status of ESMF_FIELDSTATUS_GRIDSET.

arrayspec Data type and kind specification.

[indexflag] Indicate how DE-local indices are defined. By default each DE's exclusive region is placed to start at the local index space origin, i.e. (1, 1, ..., 1). Alternatively the DE-local index space can be aligned with the global index space, if a global index space is well defined by the associated Grid. See section **??** for a list of valid indexflag options.

[gridToFieldMap] List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the field by specifying the appropriate field dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the field in sequence, i.e. gridToFieldMap = (/1,2,3,.../). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the field rank. It is erroneous to specify the same gridToFieldMap entry multiple times. The total ungridded dimensions in the field are the total field dimensions less the dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should be max(totalLWidth + totalUWidth + computationalCount, exclusiveCount).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should max(totalLWidth + totalUWidth + computationalCount, exclusiveCount).

[pinflag] Specify which type of resource DEs are pinned to. See section **??** for a list of valid pinning options. The default is to pin DEs to PETs, i.e. only the PET on which a DE was created considers the DE as local.

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

26.6.27 ESMF_FieldEmptyComplete - Complete a Field from typekind

INTERFACE:

```
   ! Private name; call using ESMF_FieldEmptyComplete()
 subroutine ESMF_FieldEmptyCompTK(field, typekind, indexflag, &
  gridToFieldMap, ungriddedLBound, ungriddedUBound, totalLWidth, totalUWidth, &
  pinflag, rc)
```

ARGUMENTS:

```
  type(ESMF_Field), intent(inout) :: field
  type(ESMF_TypeKind_Flag), intent(in) :: typekind
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_Index_Flag), intent(in), optional :: indexflag
  integer, intent(in), optional :: gridToFieldMap(:)
  integer, intent(in), optional :: ungriddedLBound(:)
  integer, intent(in), optional :: ungriddedUBound(:)
  integer, intent(in), optional :: totalLWidth(:)
  integer, intent(in), optional :: totalUWidth(:)
  type(ESMF_Pin_Flag), intent(in), optional :: pinflag
  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:

  8.1.0 Added argument `pinflag` to provide access to DE sharing between PETs.

DESCRIPTION:

Complete an `ESMF_Field` and allocate space internally for an `ESMF_Array` based on typekind. The input `ESMF_Field` must have a status of `ESMF_FIELDSTATUS_GRIDSET`. After this call the completed `ESMF_Field` has a status of `ESMF_FIELDSTATUS_COMPLETE`.

For an example and associated documentation using this method see section 26.3.7.

The arguments are:

field The input `ESMF_Field` with a status of `ESMF_FIELDSTATUS_GRIDSET`.

typekind Data type and kind specification.

[indexflag] Indicate how DE-local indices are defined. By default each DE's exclusive region is placed to start at the local index space origin, i.e. (1, 1, ..., 1). Alternatively the DE-local index space can be aligned with the global index space, if a global index space is well defined by the associated Grid. See section **??** for a list of valid indexflag options.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `field` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `field` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the field. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should be max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the Field. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[pinflag] Specify which type of resource DEs are pinned to. See section **??** for a list of valid pinning options. The default is to pin DEs to PETs, i.e. only the PET on which a DE was created considers the DE as local.

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

26.6.28 ESMF_FieldEmptyComplete - Complete a Field from Fortran array

INTERFACE:

```
! Private name; call using ESMF_FieldEmptyComplete()
subroutine ESMF_FieldEmptyComp<rank><type><kind>(field, &
farray, indexflag, datacopyflag, gridToFieldMap, &
ungriddedLBound, ungriddedUBound, totalLWidth, totalUWidth, rc)
```

ARGUMENTS:

```
  type(ESMF_Field), intent(inout) :: field
  <type> (ESMF_KIND_<kind>),intent(in), target :: farray(<rank>)
  type(ESMF_Index_Flag), intent(in) :: indexflag
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
  integer, intent(in), optional :: gridToFieldMap(:)
  integer, intent(in), optional :: ungriddedLBound(:)
  integer, intent(in), optional :: ungriddedUBound(:)
  integer, intent(in), optional :: totalLWidth(:)
  integer, intent(in), optional :: totalUWidth(:)
  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:

Complete an ESMF_Field and allocate space internally for an ESMF_Array based on typekind. The input ESMF_Field must have a status of ESMF_FIELDSTATUS_GRIDSET. After this call the completed ESMF_Field has a status of ESMF_FIELDSTATUS_COMPLETE.

The Fortran data pointer inside ESMF_Field can be queried but deallocating the retrieved data pointer is not allowed.

For an example and associated documentation using this method see section 26.3.8.

The arguments are:

field The input ESMF_Field with a status of ESMF_FIELDSTATUS_GRIDSET. The ESMF_Field will have the same dimension (dimCount) as the rank of the farray.

farray Native Fortran data array to be copied/referenced in the field. The field dimension (dimCount) will be the same as the dimCount for the farray.

indexflag Indicate how DE-local indices are defined. See section **??** for a list of valid indexflag options.

[datacopyflag] Indicates whether to copy the farray or reference it directly. For valid values see **??**. The default is ESMF_DATACOPY_REFERENCE.

[gridToFieldMap] List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the farray by specifying the appropriate farray dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the farray in sequence, i.e. gridToFieldMap = (/1,2,3,.../). Unmapped farray dimensions are undistributed Field dimensions. All gridToFieldMap entries must be greater than or equal to zero and smaller than or equal to the Field dimCount. It is erroneous to specify the same entry multiple times unless it is zero. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the `field`. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should be max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the `field`. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

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

26.6.29 ESMF_FieldEmptyComplete - Complete a Field from Fortran array pointer

INTERFACE:

```
! Private name; call using ESMF_FieldEmptyComplete()
subroutine ESMF_FieldEmptyCompPtr<rank><type><kind>(field, &
farrayPtr, datacopyflag, gridToFieldMap, &
totalLWidth, totalUWidth, rc)
```

ARGUMENTS:

```
  type(ESMF_Field), intent(inout) :: field
  <type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
  integer, intent(in), optional :: gridToFieldMap(:)
  integer, intent(in), optional :: totalLWidth(:)
  integer, intent(in), optional :: totalUWidth(:)
  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:

Complete an ESMF_Field and allocate space internally for an ESMF_Array based on typekind. The input ESMF_Field must have a status of ESMF_FIELDSTATUS_GRIDSET. After this call the completed ESMF_Field has a status of ESMF_FIELDSTATUS_COMPLETE.

The Fortran data pointer inside ESMF_Field can be queried and deallocated when datacopyflag is ESMF_DATACOPY_REFERENCE. Note that the ESMF_FieldDestroy call does not deallocate the Fortran data pointer in this case. This gives user more flexibility over memory management.

The arguments are:

field The input ESMF_Field with a status of ESMF_FIELDSTATUS_GRIDSET. The ESMF_Field will have the same dimension (dimCount) as the rank of the farrayPtr.

farrayPtr Native Fortran data pointer to be copied/referenced in the field. The field dimension (dimCount) will be the same as the dimCount for the farrayPtr.

[datacopyflag] Indicates whether to copy the farrayPtr or reference it directly. For valid values see **??**. The default is ESMF_DATACOPY_REFERENCE.

[gridToFieldMap] List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the farrayPtr by specifying the appropriate farrayPtr dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the farrayPtr in sequence, i.e. gridToFieldMap = (/1,2,3,.../). Unmapped farrayPtr dimensions are undistributed Field dimensions. All gridToFieldMap entries must be greater than or equal to zero and smaller than or equal to the Field dimCount. It is erroneous to specify the same entry multiple times unless it is zero. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the field. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should be max(totalLWidth + totalUWidth + computationalCount, exclusiveCount).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the field. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should max(totalLWidth + totalUWidth + computationalCount, exclusiveCount).

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

26.6.30 ESMF_FieldEmptyComplete - Complete a Field from Grid started with FieldEmptyCreate

INTERFACE:

```
! Private name; call using ESMF_FieldEmptyComplete()
subroutine ESMF_FieldEmptyCompGrid<rank><type><kind>(field, grid, &
farray, indexflag, datacopyflag, staggerloc, gridToFieldMap, &
ungriddedLBound, ungriddedUBound, totalLWidth, totalUWidth, rc)
```

ARGUMENTS:

```
   type(ESMF_Field), intent(inout) :: field
   type(ESMF_Grid), intent(in) :: grid
   <type> (ESMF_KIND_<kind>),intent(in), target :: farray(<rank>)
   type(ESMF_Index_Flag), intent(in) :: indexflag
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
   type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
   type(ESMF_STAGGERLOC), intent(in), optional :: staggerloc
   integer, intent(in), optional :: gridToFieldMap(:)
   integer, intent(in), optional :: ungriddedLBound(:)
   integer, intent(in), optional :: ungriddedUBound(:)
   integer, intent(in), optional :: totalLWidth(:)
   integer, intent(in), optional :: totalUWidth(:)
   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 call completes an `ESMF_Field` allocated with the `ESMF_FieldEmptyCreate()` call.

The Fortran data pointer inside `ESMF_Field` can be queried but deallocating the retrieved data pointer is not allowed.

The arguments are:

field The `ESMF_Field` object to be completed and committed in this call. The `field` will have the same dimension (dimCount) as the rank of the `farray`.

grid The `ESMF_Grid` object to complete the Field.

farray Native Fortran data array to be copied/referenced in the `field`. The `field` dimension (dimCount) will be the same as the dimCount for the `farray`.

indexflag Indicate how DE-local indices are defined. See section **??** for a list of valid indexflag options.

[datacopyflag] Indicates whether to copy the `farray` or reference it directly. For valid values see **??**. The default is `ESMF_DATACOPY_REFERENCE`.

[staggerloc] Stagger location of data in grid cells. For valid predefined values see section 31.2.6. To create a custom stagger location see section 31.3.25. The default value is `ESMF_STAGGERLOC_CENTER`.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `farray` by specifying the appropriate `farray` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `farray` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). Unmapped `farray` dimensions are undistributed Field dimensions. All `gridToFieldMap` entries must be greater than or equal to zero and smaller than or equal to the Field dimCount. It is erroneous to specify the same entry multiple times unless it is zero. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the `field`. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should be max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the `field`. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

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

26.6.31 ESMF_FieldEmptyComplete - Complete a Field from Grid started with FieldEmptyCreate

INTERFACE:

```
! Private name; call using ESMF_FieldEmptyComplete()
subroutine ESMF_FieldEmptyCompGridPtr<rank><type><kind>(field, grid, &
farrayPtr, datacopyflag, staggerloc, gridToFieldMap, &
totalLWidth, totalUWidth, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout) :: field
type(ESMF_Grid), intent(in) :: grid
<type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
type(ESMF_STAGGERLOC), intent(in), optional :: staggerloc
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: totalLWidth(:)
integer, intent(in), optional :: totalUWidth(:)
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 call completes an ESMF_Field allocated with the ESMF_FieldEmptyCreate() call.

The Fortran data pointer inside ESMF_Field can be queried and deallocated when datacopyflag is ESMF_DATACOPY_REFERENCE. Note that the ESMF_FieldDestroy call does not deallocate the Fortran data pointer in this case. This gives user more flexibility over memory management.

The Fortran data pointer inside ESMF_Field can be queried and deallocated when

The arguments are:

field The ESMF_Field object to be completed and committed in this call. The field will have the same dimension (dimCount) as the rank of the farrayPtr.

grid The ESMF_Grid object to complete the Field.

farrayPtr Native Fortran data pointer to be copied/referenced in the field. The field dimension (dimCount) will be the same as the dimCount for the farrayPtr.

[datacopyflag] Indicates whether to copy the farrayPtr or reference it directly. For valid values see **??**. The default is ESMF_DATACOPY_REFERENCE.

[staggerloc] Stagger location of data in grid cells. For valid predefined values see section 31.2.6. To create a custom stagger location see section 31.3.25. The default value is ESMF_STAGGERLOC_CENTER.

[gridToFieldMap] List with number of elements equal to the grid's dimCount. The list elements map each dimension of the grid to a dimension in the farrayPtr by specifying the appropriate farrayPtr dimension index. The default is to map all of the grid's dimensions against the lowest dimensions of the farrayPtr in sequence, i.e. gridToFieldMap = (/1,2,3,.../). Unmapped farrayPtr dimensions are undistributed Field dimensions. All gridToFieldMap entries must be greater than or equal to zero and smaller than or equal to the Field dimCount. It is erroneous to specify the same entry multiple times unless it is zero. If the Field dimCount is less than the Grid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Grid dimension will be replicating the Field across the DEs along this direction.

[totalLWidth] Lower bound of halo region. The size of this array is the number of gridded dimensions in the field. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should be max(totalLWidth + totalUWidth + computationalCount, exclusiveCount).

[totalUWidth] Upper bound of halo region. The size of this array is the number of gridded dimensions in the field. However, ordering of the elements needs to be the same as they appear in the field. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the field. That is, for each gridded dimension the field size should max(totalLWidth + totalUWidth + computationalCount, exclusiveCount).

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

26.6.32 ESMF_FieldEmptyComplete - Complete a Field from LocStream started with FieldEmptyCreate

INTERFACE:

```
! Private name; call using ESMF_FieldEmptyComplete()
subroutine ESMF_FieldEmptyCompLS<rank><type><kind>(field, locstream, &
farray, indexflag, datacopyflag, gridToFieldMap, &
ungriddedLBound, ungriddedUBound, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout) :: field
type(ESMF_LocStream), intent(in) :: locstream
<type> (ESMF_KIND_<kind>), intent(in), target :: farray(<rank>)
type(ESMF_Index_Flag), intent(in) :: indexflag
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
integer, intent(out), optional :: rc
```

DESCRIPTION:

This call completes an `ESMF_Field` allocated with the `ESMF_FieldEmptyCreate()` call.

The Fortran data pointer inside `ESMF_Field` can be queried but deallocating the retrieved data pointer is not allowed.

The arguments are:

field The `ESMF_Field` object to be completed and committed in this call. The `field` will have the same dimension (dimCount) as the rank of the `farray`.

locstream The `ESMF_LocStream` object to complete the Field.

farray Native Fortran data array to be copied/referenced in the `field`. The `field` dimension (dimCount) will be the same as the dimCount for the `farray`.

indexflag Indicate how DE-local indices are defined. See section **??** for a list of valid indexflag options.

[datacopyflag] Indicates whether to copy the `farray` or reference it directly. For valid values see **??**. The default is `ESMF_DATACOPY_REFERENCE`.

[gridToFieldMap] List with number of elements equal to the `locstream`'s dimCount. The list elements map each dimension of the `locstream` to a dimension in the `farray` by specifying the appropriate `farray` dimension index. The default is to map all of the `locstream`'s dimensions against the lowest dimensions of the `farray` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). Unmapped `farray` dimensions are undistributed Field dimensions. All `gridToFieldMap` entries must be greater than or equal to zero and smaller than or equal to the Field dimCount. It is erroneous to specify the same entry multiple times unless it is zero. If the Field dimCount is less than the LocStream dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular LocStream dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than locstream dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than locstream dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

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

26.6.33 ESMF_FieldEmptyComplete - Complete a Field from LocStream started with FieldEmptyCreate

INTERFACE:

```
! Private name; call using ESMF_FieldEmptyComplete()
subroutine ESMF_FieldEmptyCompLSPtr<rank><type><kind>(field, locstream, &
farrayPtr, datacopyflag, gridToFieldMap, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout) :: field
type(ESMF_LocStream), intent(in) :: locstream
<type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(out), optional :: rc
```

DESCRIPTION:

This call completes an `ESMF_Field` allocated with the `ESMF_FieldEmptyCreate()` call.

The Fortran data pointer inside `ESMF_Field` can be queried and deallocated when datacopyflag is `ESMF_DATACOPY_REFERENCE`. Note that the `ESMF_FieldDestroy` call does not deallocate the Fortran data pointer in this case. This gives user more flexibility over memory management.

The arguments are:

field The `ESMF_Field` object to be completed and committed in this call. The `field` will have the same dimension (dimCount) as the rank of the `farrayPtr`.

locstream The `ESMF_LocStream` object to complete the Field.

farrayPtr Native Fortran data pointer to be copied/referenced in the `field`. The `field` dimension (dimCount) will be the same as the dimCount for the `farrayPtr`.

[datacopyflag] Indicates whether to copy the `farrayPtr` or reference it directly. For valid values see **??**. The default is `ESMF_DATACOPY_REFERENCE`.

[gridToFieldMap] List with number of elements equal to the `locstream`'s dimCount. The list elements map each dimension of the `locstream` to a dimension in the `farrayPtr` by specifying the appropriate `farrayPtr` dimension index. The default is to map all of the `locstream`'s dimensions against the lowest dimensions of the `farrayPtr` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). Unmapped `farrayPtr` dimensions are undistributed Field dimensions. All `gridToFieldMap` entries must be greater than or equal to zero and smaller than or equal to the Field dimCount. It is erroneous to specify the same entry multiple times unless it is zero. If the Field dimCount is less than the LocStream dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular LocStream dimension will be replicating the Field across the DEs along this direction.

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

26.6.34 ESMF_FieldEmptyComplete - Complete a Field from Mesh started with FieldEmptyCreate

INTERFACE:

```
! Private name; call using ESMF_FieldEmptyComplete()
subroutine ESMF_FieldEmptyCompMesh<rank><type><kind>(field, mesh, &
farray, indexflag, datacopyflag, meshloc, &
gridToFieldMap, ungriddedLBound, ungriddedUBound, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout) :: field
type(ESMF_Mesh), intent(in) :: mesh
<type> (ESMF_KIND_<kind>), intent(in), target :: farray(<rank>)
type(ESMF_Index_Flag), intent(in) :: indexflag
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
type(ESMF_MeshLoc), intent(in), optional :: meshloc
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(in), optional :: ungriddedLBound(:)
integer, intent(in), optional :: ungriddedUBound(:)
integer, intent(out), optional :: rc
```

DESCRIPTION:

This call completes an `ESMF_Field` allocated with the `ESMF_FieldEmptyCreate()` call.

The Fortran data pointer inside `ESMF_Field` can be queried but deallocating the retrieved data pointer is not allowed.

The arguments are:

field The ESMF_Field object to be completed and committed in this call. The field will have the same dimension (dimCount) as the rank of the farray.

mesh The ESMF_Mesh object to complete the Field.

farray Native Fortran data array to be copied/referenced in the field. The field dimension (dimCount) will be the same as the dimCount for the farray.

indexflag Indicate how DE-local indices are defined. See section **??** for a list of valid indexflag options.

[datacopyflag] Indicates whether to copy the farray or reference it directly. For valid values see **??**. The default is ESMF_DATACOPY_REFERENCE.

[meshloc] Which part of the mesh to build the Field on. Can be set to either ESMF_MESHLOC_NODE or ESMF_MESHLOC_ELEMENT. If not set, defaults to ESMF_MESHLOC_NODE.

[gridToFieldMap] List with number of elements equal to the mesh's dimCount. The list elements map each dimension of the mesh to a dimension in the farray by specifying the appropriate farray dimension index. The default is to map all of the mesh's dimensions against the lowest dimensions of the farray in sequence, i.e. gridToFieldMap = (/1,2,3,.../). Unmapped farray dimensions are undistributed Field dimensions. All gridToFieldMap entries must be greater than or equal to zero and smaller than or equal to the Field dimCount. It is erroneous to specify the same entry multiple times unless it is zero. If the Field dimCount is less than the Mesh dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Mesh dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than Mesh dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than Mesh dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

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

26.6.35 ESMF_FieldEmptyComplete - Complete a Field from Mesh started with FieldEmptyCreate

INTERFACE:

```
! Private name; call using ESMF_FieldEmptyComplete()
subroutine ESMF_FieldEmptyCompMeshPtr<rank><type><kind>(field, mesh, &
farrayPtr, datacopyflag, meshloc, gridToFieldMap, rc)
```

ARGUMENTS:

```
  type(ESMF_Field), intent(inout) :: field
  type(ESMF_Mesh), intent(in) :: mesh
  <type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
  type(ESMF_MeshLoc), intent(in), optional :: meshloc
  integer, intent(in), optional :: gridToFieldMap(:)
  integer, intent(out), optional :: rc
```

DESCRIPTION:

This call completes an ESMF_Field allocated with the ESMF_FieldEmptyCreate() call.

The Fortran data pointer inside ESMF_Field can be queried and deallocated when datacopyflag is ESMF_DATACOPY_REFERENCE. Note that the ESMF_FieldDestroy call does not deallocate the Fortran data pointer in this case. This gives user more flexibility over memory management.

The arguments are:

field The ESMF_Field object to be completed and committed in this call. The field will have the same dimension (dimCount) as the rank of the farrayPtr.

mesh The ESMF_Mesh object to complete the Field.

farrayPtr Native Fortran data pointer to be copied/referenced in the field. The field dimension (dimCount) will be the same as the dimCount for the farrayPtr.

[datacopyflag] Indicates whether to copy the farrayPtr or reference it directly. For valid values see **??**. The default is ESMF_DATACOPY_REFERENCE.

[meshloc] Which part of the mesh to build the Field on. Can be set to either ESMF_MESHLOC_NODE or ESMF_MESHLOC_ELEMENT. If not set, defaults to ESMF_MESHLOC_NODE.

[gridToFieldMap] List with number of elements equal to the mesh's dimCount. The list elements map each dimension of the mesh to a dimension in the farrayPtr by specifying the appropriate farrayPtr dimension index. The default is to map all of the mesh's dimensions against the lowest dimensions of the farrayPtr in sequence, i.e. gridToFieldMap = (/1,2,3,.../). Unmapped farrayPtr dimensions are undistributed Field dimensions. All gridToFieldMap entries must be greater than or equal to zero and smaller than or equal to the Field dimCount. It is erroneous to specify the same entry multiple times unless it is zero. If the Field dimCount is less than the Mesh dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the gridToFieldMap indicates that the particular Mesh dimension will be replicating the Field across the DEs along this direction.

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

26.6.36 ESMF_FieldEmptyComplete - Complete a Field from XGrid started with FieldEmptyCreate

INTERFACE:

```
  ! Private name; call using ESMF_FieldEmptyComplete()
  subroutine ESMF_FieldEmptyCompXG<rank><type><kind>(field, xgrid, &
```

```
   farray, indexflag, datacopyflag, xgridside, gridindex, &
   gridToFieldMap, &
   ungriddedLBound, ungriddedUBound, rc)
```

ARGUMENTS:

```
   type(ESMF_Field), intent(inout) :: field
   type(ESMF_XGrid), intent(in) :: xgrid
   <type> (ESMF_KIND_<kind>), intent(in), target :: farray(<rank>)
   type(ESMF_Index_Flag), intent(in) :: indexflag
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
   type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
   type(ESMF_XGridSide_Flag), intent(in), optional :: xgridside
   integer, intent(in), optional :: gridindex
   integer, intent(in), optional :: gridToFieldMap(:)
   integer, intent(in), optional :: ungriddedLBound(:)
   integer, intent(in), optional :: ungriddedUBound(:)
   integer, intent(out), optional :: rc
```

DESCRIPTION:

This call completes an `ESMF_Field` allocated with the `ESMF_FieldEmptyCreate()` call.

The Fortran data pointer inside `ESMF_Field` can be queried but deallocating the retrieved data pointer is not allowed.

The arguments are:

field The `ESMF_Field` object to be completed and committed in this call. The `field` will have the same dimension (dimCount) as the rank of the `farray`.

xgrid The `ESMF_XGrid` object to complete the Field.

farray Native Fortran data array to be copied/referenced in the `field`. The `field` dimension (dimCount) will be the same as the dimCount for the `farray`.

indexflag Indicate how DE-local indices are defined. See section **??** for a list of valid indexflag options.

[datacopyflag] Indicates whether to copy the `farray` or reference it directly. For valid values see **??**. The default is `ESMF_DATACOPY_REFERENCE`.

[xgridside] Which side of the XGrid to create the Field on (either ESMF_XGRIDSIDE_A, ESMF_XGRIDSIDE_B, or ESMF_XGRIDSIDE_BALANCED). If not passed in then defaults to ESMF_XGRIDSIDE_BALANCED.

[gridindex] If xgridSide is ESMF_XGRIDSIDE_A or ESMF_XGRIDSIDE_B then this index tells which Grid on that side to create the Field on. If not provided, defaults to 1.

[gridToFieldMap] List with number of elements equal to the `xgrid`'s dimCount. The list elements map each dimension of the `xgrid` to a dimension in the `farray` by specifying the appropriate `farray` dimension index. The default is to map all of the `xgrid`'s dimensions against the lowest dimensions of the `farray` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). Unmapped `farray` dimensions are undistributed Field dimensions. All `gridToFieldMap` entries must be greater than or equal to zero and smaller than or equal to the Field dimCount. It is erroneous to specify the same entry multiple times unless it is zero. If the Field dimCount is less than the XGrid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular XGrid dimension will be replicating the Field across the DEs along this direction.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than XGrid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than XGrid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

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

26.6.37 ESMF_FieldEmptyComplete - Complete a Field from XGrid started with FieldEmptyCreate

INTERFACE:

```
! Private name; call using ESMF_FieldEmptyComplete()
subroutine ESMF_FieldEmptyCompXGPtr<rank><type><kind>(field, xgrid, &
farrayPtr, xgridside, gridindex, &
datacopyflag, gridToFieldMap, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout) :: field
type(ESMF_XGrid), intent(in) :: xgrid
<type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
type(ESMF_XGridSide_Flag), intent(in), optional :: xgridside
integer, intent(in), optional :: gridindex
integer, intent(in), optional :: gridToFieldMap(:)
integer, intent(out), optional :: rc
```

DESCRIPTION:

This call completes an `ESMF_Field` allocated with the `ESMF_FieldEmptyCreate()` call.

The Fortran data pointer inside `ESMF_Field` can be queried and deallocated when datacopyflag is `ESMF_DATACOPY_REFERENCE`. Note that the `ESMF_FieldDestroy` call does not deallocate the Fortran data pointer in this case. This gives user more flexibility over memory management.

The arguments are:

field The `ESMF_Field` object to be completed and committed in this call. The `field` will have the same dimension (dimCount) as the rank of the `farrayPtr`.

xgrid The `ESMF_XGrid` object to complete the Field.

farrayPtr Native Fortran data pointer to be copied/referenced in the `field`. The `field` dimension (dimCount) will be the same as the dimCount for the `farrayPtr`.

[datacopyflag] Indicates whether to copy the `farrayPtr` or reference it directly. For valid values see **??**. The default is `ESMF_DATACOPY_REFERENCE`.

[xgridside] Which side of the XGrid to create the Field on (either ESMF_XGRIDSIDE_A, ESMF_XGRIDSIDE_B, or ESMF_XGRIDSIDE_BALANCED). If not passed in then defaults to ESMF_XGRIDSIDE_BALANCED.

[gridindex] If xgridside is ESMF_XGRIDSIDE_A or ESMF_XGRIDSIDE_B then this index tells which Grid on that side to create the Field on. If not provided, defaults to 1.

[gridToFieldMap] List with number of elements equal to the `xgrid`'s dimCount. The list elements map each dimension of the `xgrid` to a dimension in the `farrayPtr` by specifying the appropriate `farrayPtr` dimension index. The default is to map all of the `xgrid`'s dimensions against the lowest dimensions of the `farrayPtr` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). Unmapped `farrayPtr` dimensions are undistributed Field dimensions. All `gridToFieldMap` entries must be greater than or equal to zero and smaller than or equal to the Field dimCount. It is erroneous to specify the same entry multiple times unless it is zero. If the Field dimCount is less than the XGrid dimCount then the default gridToFieldMap will contain zeros for the rightmost entries. A zero entry in the `gridToFieldMap` indicates that the particular XGrid dimension will be replicating the Field across the DEs along this direction.

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

26.6.38 ESMF_FieldEmptyCreate - Create an empty Field

INTERFACE:

```
function ESMF_FieldEmptyCreate(name, vm, rc)
```

RETURN VALUE:

```
type(ESMF_Field) :: ESMF_FieldEmptyCreate
```

ARGUMENTS:

```
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
 character (len = *), intent(in), optional :: name
 type(ESMF_VM), intent(in), 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:

8.0.0 Added argument vm to support object creation on a different VM than that of the current context.

DESCRIPTION:

This version of ESMF_FieldCreate builds an empty ESMF_Field and depends on later calls to add an ESMF_Grid and ESMF_Array to it. The empty ESMF_Field can be completed in one more step or two more steps by the ESMF_FieldEmptySet and ESMF_FieldEmptyComplete methods. Attributes can be added to an empty Field object. For an example and associated documentation using this method see section 26.3.8 and 26.3.7.

The arguments are:

[name] Field name.

[vm] If present, the Field object is created on the specified ESMF_VM object. The default is to create on the VM of the current component context.

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

26.6.39 ESMF_FieldEmptySet - Set a Grid in an empty Field

INTERFACE:

```
! Private name; call using ESMF_FieldEmptySet()
subroutine ESMF_FieldEmptySetGrid(field, grid, StaggerLoc, &
  vm, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout) :: field
type(ESMF_Grid), intent(in) :: grid
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_STAGGERLOC), intent(in), optional :: StaggerLoc
type(ESMF_VM), intent(in), 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:

  7.1.0r Added argument vm to support object creation on a different VM than that of the current context.

DESCRIPTION:

Set a grid and an optional staggerloc (default to center stagger ESMF_STAGGERLOC_CENTER) in a non-completed ESMF_Field. The ESMF_Field must not be completed for this to succeed. After this operation, the ESMF_Field contains the ESMF_Grid internally but holds no data. The status of the field changes from ESMF_FIELDSTATUS_EMPTY to ESMF_FIELDSTATUS_GRIDSET or stays ESMF_FIELDSTATUS_GRIDSET.

For an example and associated documentation using this method see section 26.3.7.

The arguments are:

field Empty ESMF_Field. After this operation, the ESMF_Field contains the ESMF_Grid internally but holds no data. The status of the field changes from ESMF_FIELDSTATUS_EMPTY to ESMF_FIELDSTATUS_GRIDSET.

grid ESMF_Grid to be set in the ESMF_Field.

[StaggerLoc] Stagger location of data in grid cells. For valid predefined values see section 31.2.6. To create a custom stagger location see section 31.3.25. The default value is ESMF_STAGGERLOC_CENTER.

[vm] If present, the Field object will only be accessed, and the Grid object set, on those PETs contained in the specified ESMF_VM object. The default is to assume the VM of the current context.

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

26.6.40 ESMF_FieldEmptySet - Set a Mesh in an empty Field

INTERFACE:

```
! Private name; call using ESMF_FieldEmptySet()
subroutine ESMF_FieldEmptySetMesh(field, mesh, indexflag, meshloc, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout) :: field
type(ESMF_Mesh), intent(in) :: mesh
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_Index_Flag),intent(in), optional :: indexflag
type(ESMF_MeshLoc), intent(in), optional :: meshloc
integer, intent(out), optional :: rc
```

DESCRIPTION:

Set a mesh and an optional meshloc (default to center stagger ESMF_MESHLOC_NODE) in a non-completed ESMF_Field. The ESMF_Field must not be completed for this to succeed. After this operation, the ESMF_Field contains the ESMF_Mesh internally but holds no data. The status of the field changes from ESMF_FIELDSTATUS_EMPTY to ESMF_FIELDSTATUS_GRIDSET or stays ESMF_FIELDSTATUS_GRIDSET.

The arguments are:

field Empty ESMF_Field. After this operation, the ESMF_Field contains the ESMF_Mesh internally but holds no data. The status of the field changes from ESMF_FIELDSTATUS_EMPTY to ESMF_FIELDSTATUS_GRIDSET.

mesh ESMF_Mesh to be set in the ESMF_Field.

[indexflag] Indicate how DE-local indices are defined. See section **??** for a list of valid indexflag options.

[meshloc] Which part of the mesh to build the Field on. Can be set to either ESMF_MESHLOC_NODE or ESMF_MESHLOC_ELEMENT. If not set, defaults to ESMF_MESHLOC_NODE.

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

26.6.41 ESMF_FieldEmptySet - Set a LocStream in an empty Field

INTERFACE:

```
! Private name; call using ESMF_FieldEmptySet()
subroutine ESMF_FieldEmptySetLocStream(field, locstream, &
  vm, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout) :: field
type(ESMF_LocStream), intent(in) :: locstream
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_VM), intent(in), optional :: vm
integer, intent(out), optional :: rc
```

DESCRIPTION:

Set a ESMF_LocStream in a non-completed ESMF_Field. The ESMF_Field must not be completed for this to succeed. After this operation, the ESMF_Field contains the ESMF_LocStream internally but holds no data. The status of the field changes from ESMF_FIELDSTATUS_EMPTY to ESMF_FIELDSTATUS_GRIDSET or stays ESMF_FIELDSTATUS_GRIDSET.

The arguments are:

field Empty ESMF_Field. After this operation, the ESMF_Field contains the ESMF_LocStream internally but holds no data. The status of the field changes from ESMF_FIELDSTATUS_EMPTY to ESMF_FIELDSTATUS_GRIDSET.

locstream ESMF_LocStream to be set in the ESMF_Field.

[vm] If present, the Field object will only be accessed, and the Grid object set, on those PETs contained in the specified ESMF_VM object. The default is to assume the VM of the current context.

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

26.6.42 ESMF_FieldEmptySet - Set an XGrid in an empty Field

INTERFACE:

```
! Private name; call using ESMF_FieldEmptySet()
subroutine ESMF_FieldEmptySetXGrid(field, xgrid, xgridside, gridindex, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout) :: field
type(ESMF_XGrid), intent(in) :: xgrid
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_XGridSide_Flag), intent(in), optional :: xgridside
integer, intent(in), optional :: gridindex
integer, intent(out), optional :: rc
```

DESCRIPTION:

Set a xgrid and optional xgridside (default to balanced side ESMF_XGRIDSIDE_Balanced) and gridindex (default to 1) in a non-complete ESMF_Field. The ESMF_Field must not be completed for this to succeed. After this operation, the ESMF_Field contains the ESMF_XGrid internally but holds no data. The status of the field changes from ESMF_FIELDSTATUS_EMPTY to ESMF_FIELDSTATUS_GRIDSET or stays ESMF_FIELDSTATUS_GRIDSET.

The arguments are:

field Empty ESMF_Field. After this operation, the ESMF_Field contains the ESMF_XGrid internally but holds no data. The status of the field changes from ESMF_FIELDSTATUS_EMPTY to ESMF_FIELDSTATUS_GRIDSET.

xgrid ESMF_XGrid to be set in the ESMF_Field.

[xgridside] Side of XGrid to retrieve a DistGrid. For valid predefined values see section **??**. The default value is ESMF_XGRIDSIDE_BALANCED.

[gridindex] Index to specify which DistGrid when on side A or side B. The default value is 1.

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

26.6.43 ESMF_FieldFill - Fill data into a Field

INTERFACE:

```
subroutine ESMF_FieldFill(field, dataFillScheme, &
  const1, member, step, &
  param1I4, param2I4, param3I4, &
  param1R4, param2R4, param3R4, &
  param1R8, param2R8, param3R8, &
  rc)
```

ARGUMENTS:

```
    type(ESMF_Field), intent(inout) :: field
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    character(len=*), intent(in), optional :: dataFillScheme
    real(ESMF_KIND_R8), intent(in), optional :: const1
    integer, intent(in), optional :: member
    integer, intent(in), optional :: step
    integer(ESMF_KIND_I4), intent(in), optional :: param1I4
    integer(ESMF_KIND_I4), intent(in), optional :: param2I4
    integer(ESMF_KIND_I4), intent(in), optional :: param3I4
    real(ESMF_KIND_R4), intent(in), optional :: param1R4
    real(ESMF_KIND_R4), intent(in), optional :: param2R4
    real(ESMF_KIND_R4), intent(in), optional :: param3R4
    real(ESMF_KIND_R8), intent(in), optional :: param1R8
    real(ESMF_KIND_R8), intent(in), optional :: param2R8
    real(ESMF_KIND_R8), intent(in), optional :: param3R8
    integer, intent(out), optional :: rc
```

DESCRIPTION:

Fill `field` with data according to `dataFillScheme`. Depending on the chosen fill scheme, the `member` and `step` arguments are used to provide differing fill data patterns.

The arguments are:

field The `ESMF_Field` object to fill with data.

[dataFillScheme] The fill scheme. The available options are "sincos", "one", and "const". Defaults to "sincos".

[const1] Constant of real type. Defaults to 0.

[member] Member incrementor. Defaults to 1.

[step] Step incrementor. Defaults to 1.

[param1I4] Optional parameter of typekind I4. The default depends on the specified `dataFillScheme`.

[param2I4] Optional parameter of typekind I4. The default depends on the specified `dataFillScheme`.

[param3I4] Optional parameter of typekind I4. The default depends on the specified `dataFillScheme`.

[param1R4] Optional parameter of typekind R4. The default depends on the specified `dataFillScheme`.

[param2R4] Optional parameter of typekind R4. The default depends on the specified `dataFillScheme`.

[param3R4] Optional parameter of typekind R4. The default depends on the specified `dataFillScheme`.

[param1R8] Optional parameter of typekind R8. The default depends on the specified `dataFillScheme`.

[param2R8] Optional parameter of typekind R8. The default depends on the specified `dataFillScheme`.

[param3R8] Optional parameter of typekind R8. The default depends on the specified `dataFillScheme`.

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

26.6.44 ESMF_FieldGather - Gather a Fortran array from an ESMF_Field

INTERFACE:

```
subroutine ESMF_FieldGather<rank><type><kind>(field, farray, &
rootPet, tile, vm, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in) :: field
<type>(ESMF_KIND_<kind>), intent(out), target :: farray(<rank>)
integer, intent(in) :: rootPet
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
integer, intent(in), optional :: tile
type(ESMF_VM), intent(in), 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.

DESCRIPTION:

Gather the data of an ESMF_Field object into the `farray` located on `rootPET`. A single DistGrid tile of `array` must be gathered into `farray`. The optional `tile` argument allows selection of the tile. For Fields defined on a single tile DistGrid the default selection (tile 1) will be correct. The shape of `farray` must match the shape of the tile in Field.

If the Field contains replicating DistGrid dimensions data will be gathered from the numerically higher DEs. Replicated data elements in numericaly lower DEs will be ignored.

The implementation of Scatter and Gather is not sequence index based. If the Field is built on arbitrarily distributed Grid, Mesh, LocStream or XGrid, Gather will not gather data to rootPet from source data points corresponding to the sequence index on rootPet. Instead Gather will gather a contiguous memory range from source PET to rootPet. The size of the memory range is equal to the number of data elements on the source PET. Vice versa for the Scatter operation. In this case, the user should use `ESMF_FieldRedist` to achieve the same data operation result. For examples how to use `ESMF_FieldRedist` to perform Gather and Scatter, please refer to 26.3.32 and 26.3.31.

This version of the interface implements the PET-based blocking paradigm: Each PET of the VM must issue this call exactly once for *all* of its DEs. The call will block until all PET-local data objects are accessible.

For examples and associated documentation regarding this method see Section 26.3.28.

The arguments are:

field The `ESMF_Field` object from which data will be gathered.

{farray} The Fortran array into which to gather data. Only root must provide a valid `farray`, the other PETs may treat `farray` as an optional argument.

rootPet PET that holds the valid destination array, i.e. `farray`.

[tile] The DistGrid tile in `field` from which to gather `farray`. By default `farray` will be gathered from tile 1.

[vm] Optional `ESMF_VM` object of the current context. Providing the VM of the current context will lower the method's overhead.

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

26.6.45 ESMF_FieldGet - Get object-wide Field information

INTERFACE:

```
! Private name; call using ESMF_FieldGet()
subroutine ESMF_FieldGetDefault(field, arrayspec, &
  status, geomtype, grid, mesh, locstream, xgrid, array, localarrayList, &
  typekind, dimCount, rank, staggerloc, meshloc, xgridside, &
  gridindex, gridToFieldMap, ungriddedLBound, ungriddedUBound, &
  totalLWidth, totalUWidth, localDeCount, ssiLocalDeCount, &
  localDeToDeMap, minIndex, maxIndex, elementCount, &
  localMinIndex, localMaxIndex, localElementCount, name, vm, rc)
```

ARGUMENTS:

```
  type(ESMF_Field), intent(in) :: field
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_ArraySpec), intent(out), optional :: arrayspec
  type(ESMF_FieldStatus_Flag),intent(out), optional :: status
  type(ESMF_GeomType_Flag), intent(out), optional :: geomtype
  type(ESMF_Grid), intent(out), optional :: grid
  type(ESMF_Mesh), intent(out), optional :: mesh
  type(ESMF_LocStream), intent(out), optional :: locstream
  type(ESMF_XGrid), intent(out), optional :: xgrid
  type(ESMF_Array), intent(out), optional :: array
  type(ESMF_LocalArray), target, intent(out), optional :: localarrayList(:)
  type(ESMF_TypeKind_Flag), intent(out), optional :: typekind
  integer, intent(out), optional :: dimCount
  integer, intent(out), optional :: rank
  type(ESMF_StaggerLoc), intent(out), optional :: staggerloc
  type(ESMF_MeshLoc), intent(out), optional :: meshloc
  type(ESMF_XGridSide_Flag), intent(out), optional :: xgridside
  integer, intent(out), optional :: gridindex
  integer, intent(out), optional :: gridToFieldMap(:)
  integer, intent(out), optional :: ungriddedLBound(:)
  integer, intent(out), optional :: ungriddedUBound(:)
  integer, intent(out), optional :: totalLWidth(:,:)
  integer, intent(out), optional :: totalUWidth(:,:)
  integer, intent(out), optional :: localDeCount
  integer, intent(out), optional :: ssiLocalDeCount
  integer, intent(out), optional :: localDeToDeMap(:)
  integer, intent(out), optional :: minIndex(:)
  integer, intent(out), optional :: maxIndex(:)
  integer, intent(out), optional :: elementCount(:)
```

```
integer, intent(out), optional :: localMinIndex(:)
integer, intent(out), optional :: localMaxIndex(:)
integer, intent(out), optional :: localElementCount(:)
character(len=*), intent(out), optional :: name
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 - *except those arguments indicated below*.
- 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:

  6.3.0r Added argument `vm` in order to offer information about the VM on which the Field was created.

  8.1.0 Added argument `minIndex`. The new argument allows the user to query the global lower bounds of the field data across all PETs.
  Added argument `maxIndex`. The new argument allows the user to query the global upper bounds of the field data across all PETs.
  Added argument `elementCount`. The new argument allows the user to query the global number of items of the field data across all PETs.
  Added argument `localMinIndex`. The new argument allows the user to query the PET local lower bounds globally indexed of the field data.
  Added argument `localMaxIndex`. The new argument allows the user to query the PET local upper bounds globally indexed of the field data.
  Added argument `localElementCount`. The new argument allows the user to query the PET local number of items of the field data.
  Added argument `ssiLocalDeCount` and `localarrayList` to support DE sharing between PETs on the same single system image (SSI).
  Added argument `localDeToDeMap` to support DE handling from the Field level rather than require user to go to Array level.

DESCRIPTION:

Query an `ESMF_Field` object for various pieces of information. All arguments after the `field` argument are optional. To select individual items use the `named_argument=value` syntax. For an example and associated documentation using this method see section 26.3.3.

The arguments are:

field `ESMF_Field` object to query.

[arrayspec] `ESMF_ArraySpec` object containing the type/kind/rank information of the Field object.

[status] The status of the Field. See section 26.2.1 for a complete list of values.

[geomtype] The type of geometry on which the Field is built. See section **??** for the range of values.

[grid] `ESMF_Grid`.

[mesh] STATUS:*This argument is excluded from the backward compatibility statement*.
`ESMF_Mesh`.

[locstream] STATUS:*This argument is excluded from the backward compatibility statement.*
`ESMF_LocStream`.

[xgrid] STATUS:*This argument is excluded from the backward compatibility statement.*
`ESMF_XGrid`.

[array] `ESMF_Array`.

[localarrayList] Upon return this holds a list of the associated `ESMC_LocalArray` objects. `localarrayList` must be allocated to be of size `localDeCount` or `ssiLocalDeCount`.

[typekind] TypeKind specifier for Field. See section **??** for a complete list of values.

[dimCount] Number of geometrical dimensions in `field`. For an detailed discussion of this parameter, please see section 26.3.23 and section 26.3.24.

[rank] Number of dimensions in the physical memory of the `field` data. It is identical to dimCount when the corresponding grid is a non-arbitrary grid. It is less than dimCount when the grid is arbitrarily distributed. For an detailed discussion of this parameter, please see section 26.3.23 and section 26.3.24.

[staggerloc] Stagger location of data in grid cells. For valid predefined values and interpretation of results see section 31.2.6.

[meshloc] STATUS:*This argument is excluded from the backward compatibility statement.*
The part of the mesh to build the Field on. Can be either `ESMF_MESHLOC_NODE` or `ESMF_MESHLOC_ELEMENT`. If not set, defaults to `ESMF_MESHLOC_NODE`.

[xgridside] STATUS:*This argument is excluded from the backward compatibility statement.*
The side of the XGrid that the Field was created on. See section **??** for a complete list of values.

[gridIndex] STATUS:*This argument is excluded from the backward compatibility statement.*
If xgridside is `ESMF_XGRIDSIDE_A` or `ESMF_XGRIDSIDE_B` then this index tells which Grid/Mesh on that side the Field was created on.

[gridToFieldMap] List with number of elements equal to the `grid`'s dimCount. The list elements map each dimension of the `grid` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map all of the `grid`'s dimensions against the lowest dimensions of the `field` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the field.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[totalLWidth] Lower bound of halo region. The size of the first dimension of this array is the number of gridded dimensions in the `field`. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should be max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`). The size of the 2nd dimension of this array is localDeCount.

[totalUWidth] Upper bound of halo region. The size of the first dimension of this array is the number of gridded dimensions in the `field`. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`). The size of the 2nd dimension of this array is localDeCount.

[localDeCount] Upon return this holds the number of PET-local DEs defined in the DELayout associated with the Field object.

[ssiLocalDeCount] The number of DEs in the Field available to the local PET. This includes DEs that are local to other PETs on the same SSI, that are accessible via shared memory.

[localDeToDeMap] Mapping between localDe indices and the (global) DEs associated with the local PET. The localDe index variables are discussed in sections **??** and 28.2.5. The provided actual argument must be of size `localDeCount`, or `ssiLocalDeCount`, and will be filled accordingly.

[minIndex] Upon return this holds the global lower bounds of the field data across all PETs. This information will be identical across all PETs. `minIndex` must be allocated to be of size equal to the field rank.

[maxIndex] Upon return this holds the global upper bounds of the field data across all PETs. This information will be identical across all PETs. `maxIndex` must be allocated to be of size equal to the field rank.

[elementCount] Upon return this holds the global number of items of the field data across all PETs. This information will be identical across all PETs. `elementCount` must be allocated to be of size equal to the field rank.

[localMinIndex] Upon return this holds the PET local lower bounds globally indexed of the field data. `localMinIndex` must be allocated to be of size equal to the field rank.

[localMaxIndex] Upon return this holds the PET local upper bounds globally indexed of the field data. `localMaxIndex` must be allocated to be of size equal to the field rank.

[localElementCount] Upon return this holds the PET local number of items of the field data. `localElementCount` must be allocated to be of size equal to the field rank.

[name] Name of queried item.

[vm] The VM on which the Field object was created.

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

26.6.46 ESMF_FieldGet - Get a DE-local Fortran array pointer from a Field

INTERFACE:

```
! Private name; call using ESMF_FieldGet()
subroutine ESMF_FieldGetDataPtr<rank><type><kind>(field, localDe, &
farrayPtr, exclusiveLBound, exclusiveUBound, exclusiveCount, &
computationalLBound, computationalUBound, computationalCount, &
totalLBound, totalUBound, totalCount, rc)
```

ARGUMENTS:

```
  type(ESMF_Field), intent(in) :: field
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  integer, intent(in), optional :: localDe
  <type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
  integer, intent(out), optional :: exclusiveLBound(:)
  integer, intent(out), optional :: exclusiveUBound(:)
  integer, intent(out), optional :: exclusiveCount(:)
  integer, intent(out), optional :: computationalLBound(:)
  integer, intent(out), optional :: computationalUBound(:)
  integer, intent(out), optional :: computationalCount(:)
  integer, intent(out), optional :: totalLBound(:)
  integer, intent(out), optional :: totalUBound(:)
  integer, intent(out), optional :: totalCount(:)
  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:

Get a Fortran pointer to DE-local memory allocation within `field`. For convenience DE-local bounds can be queried at the same time. For an example and associated documentation using this method see section 26.3.2.

The arguments are:

field `ESMF_Field` object.

[localDe] Local DE for which information is requested. `[0,..,localDeCount-1]`. For `localDeCount==1` the `localDe` argument may be omitted, in which case it will default to `localDe=0`.

farrayPtr Fortran array pointer which will be pointed at DE-local memory allocation. It depends on the specific entry point of `ESMF_FieldCreate()` used during `field` creation, which Fortran operations are supported on the returned `farrayPtr`. See 26.4 for more details.

[exclusiveLBound] Upon return this holds the lower bounds of the exclusive region. `exclusiveLBound` must be allocated to be of size equal to `field`'s `dimCount`. See section 28.2.6 for a description of the regions and their associated bounds and counts.

[exclusiveUBound] Upon return this holds the upper bounds of the exclusive region. `exclusiveUBound` must be allocated to be of size equal to `field`'s `dimCount`. See section 28.2.6 for a description of the regions and their associated bounds and counts.

[exclusiveCount] Upon return this holds the number of items, `exclusiveUBound-exclusiveLBound+1`, in the exclusive region per dimension. `exclusiveCount` must be allocated to be of size equal to `field`'s `dimCount`. See section 28.2.6 for a description of the regions and their associated bounds and counts.

[computationalLBound] Upon return this holds the lower bounds of the computational region. `computationalLBound` must be allocated to be of size equal to `field`'s `dimCount`. See section 28.2.6 for a description of the regions and their associated bounds and counts.

[computationalUBound] Upon return this holds the lower bounds of the computational region. `computationalUBound` must be allocated to be of size equal to `field`'s `dimCount`. See section 28.2.6 for a description of the regions and their associated bounds and counts.

[computationalCount] Upon return this holds the number of items in the computational region per dimension (i.e. `computationalUBound-computationalLBound+1`). `computationalCount` must be allocated to be of size equal to `field`'s `dimCount`. See section 28.2.6 for a description of the regions and their associated bounds and counts.

[totalLBound] Upon return this holds the lower bounds of the total region. `totalLBound` must be allocated to be of size equal to `field`'s `dimCount`. See section 28.2.6 for a description of the regions and their associated bounds and counts.

[totalUBound] Upon return this holds the lower bounds of the total region. `totalUBound` must be allocated to be of size equal to `field`'s `dimCount`. See section 28.2.6 for a description of the regions and their associated bounds and counts.

[totalCount] Upon return this holds the number of items in the total region per dimension (i.e. `totalUBound-totalLBound+1`). `computationalCount` must be allocated to be of size equal to `field`'s `dimCount`. See section 28.2.6 for a description of the regions and their associated bounds and counts.

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

26.6.47 ESMF_FieldGetBounds - Get DE-local Field data bounds

INTERFACE:

```
! Private name; call using ESMF_FieldGetBounds()
subroutine ESMF_FieldGetBounds(field, localDe, &
  exclusiveLBound, exclusiveUBound, exclusiveCount, computationalLBound, &
  computationalUBound, computationalCount, totalLBound, &
  totalUBound, totalCount, rc)
```

ARGUMENTS:

```
  type(ESMF_Field), intent(in) :: field
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  integer, intent(in), optional :: localDe
  integer, intent(out), optional :: exclusiveLBound(:)
  integer, intent(out), optional :: exclusiveUBound(:)
  integer, intent(out), optional :: exclusiveCount(:)
  integer, intent(out), optional :: computationalLBound(:)
  integer, intent(out), optional :: computationalUBound(:)
  integer, intent(out), optional :: computationalCount(:)
  integer, intent(out), optional :: totalLBound(:)
  integer, intent(out), optional :: totalUBound(:)
  integer, intent(out), optional :: totalCount(:)
  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 method returns the bounds information of a field that consists of a internal grid and a internal array. The exclusive and computational bounds are shared between the grid and the array but the total bounds are the array bounds plus the halo width. The count is the number of elements between each bound pair.

The arguments are:

field Field to get the information from.

[localDe] Local DE for which information is requested. `[0,..,localDeCount-1]`. For `localDeCount==1` the `localDe` argument may be omitted, in which case it will default to `localDe=0`.

[exclusiveLBound] Upon return this holds the lower bounds of the exclusive region. `exclusiveLBound` must be allocated to be of size equal to the field rank. Please see section 31.3.19 for a description of the regions and their associated bounds and counts.

[exclusiveUBound] Upon return this holds the upper bounds of the exclusive region. `exclusiveUBound` must be allocated to be of size equal to the field rank. Please see section 31.3.19 for a description of the regions and their associated bounds and counts.

[exclusiveCount] Upon return this holds the number of items, `exclusiveUBound-exclusiveLBound+1`, in the exclusive region per dimension. `exclusiveCount` must be allocated to be of size equal to the field rank. Please see section 31.3.19 for a description of the regions and their associated bounds and counts.

[computationalLBound] Upon return this holds the lower bounds of the stagger region. `computationalLBound` must be allocated to be of size equal to the field rank. Please see section 31.3.19 for a description of the regions and their associated bounds and counts.

[computationalUBound] Upon return this holds the upper bounds of the stagger region. `computationalUBound` must be allocated to be of size equal to the field rank. Please see section 31.3.19 for a description of the regions and their associated bounds and counts.

[computationalCount] Upon return this holds the number of items in the computational region per dimension (i.e. `computationalUBound-computationalLBound+1`). `computationalCount` must be allocated to be of size equal to the field rank. Please see section 31.3.19 for a description of the regions and their associated bounds and counts.

[totalLBound] Upon return this holds the lower bounds of the total region. `totalLBound` must be allocated to be of size equal to the field rank.

[totalUBound] Upon return this holds the upper bounds of the total region. `totalUBound` must be allocated to be of size equal to the field rank.

[totalCount] Upon return this holds the number of items in the total region per dimension (i.e. `totalUBound-totalLBound+1`). `totalCount` must be allocated to be of size equal to the field rank.

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

26.6.48 ESMF_FieldHalo - Execute a FieldHalo operation

INTERFACE:

```
subroutine ESMF_FieldHalo(field, routehandle,  &
                          routesyncflag, finishedflag, checkflag, rc)
```

ARGUMENTS:

```
     type(ESMF_Field),          intent(inout)          :: field
     type(ESMF_RouteHandle),    intent(inout)          :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_RouteSync_Flag), intent(in),  optional  :: routesyncflag
     logical,                   intent(out), optional  :: finishedflag
     logical,                   intent(in),  optional  :: checkflag
     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:

Execute a precomputed Field halo operation for `field`. The `field` argument must match the Field used during `ESMF_FieldHaloStore()` in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

See `ESMF_FieldHaloStore()` on how to precompute `routehandle`.

This call is *collective* across the current VM.

field `ESMF_Field` containing data to be haloed.

routehandle Handle to the precomputed Route.

[routesyncflag] Indicate communication option. Default is `ESMF_ROUTESYNC_BLOCKING`, resulting in a blocking operation. See section **??** for a complete list of valid settings.

[finishedflag] Used in combination with `routesyncflag = ESMF_ROUTESYNC_NBTESTFINISH`. Returned `finishedflag` equal to `.true.` indicates that all operations have finished. A value of `.false.` indicates that there are still unfinished operations that require additional calls with `routesyncflag = ESMF_ROUTESYNC_NBTESTFINISH`, or a final call with `routesyncflag = ESMF_ROUTESYNC_NBWAITFINISH`. For all other `routesyncflag` settings the returned value in `finishedflag` is always `.true.`.

[checkflag] If set to `.TRUE.` the input Field pair will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

26.6.49 ESMF_FieldHaloRelease - Release resources associated with a Field halo operation

INTERFACE:

```
subroutine ESMF_FieldHaloRelease(routehandle, noGarbage, rc)
```

ARGUMENTS:

```
     type(ESMF_RouteHandle), intent(inout)           :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),   optional  :: noGarbage
     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:

  8.0.0 Added argument `noGarbage`. The argument provides a mechanism to override the default garbage collection mechanism when destroying an ESMF object.

DESCRIPTION:

Release resources associated with a Field halo operation. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

[noGarbage] If set to `.TRUE.` the object will be fully destroyed and removed from the ESMF garbage collection system. Note however that under this condition ESMF cannot protect against accessing the destroyed object through dangling aliases – a situation which may lead to hard to debug application crashes.

It is generally recommended to leave the `noGarbage` argument set to `.FALSE.` (the default), and to take advantage of the ESMF garbage collection system which will prevent problems with dangling aliases or incorrect sequences of destroy calls. However this level of support requires that a small remnant of the object is kept in memory past the destroy call. This can lead to an unexpected increase in memory consumption over the course of execution in applications that use temporary ESMF objects. For situations where the repeated creation and destruction of temporary objects leads to memory issues, it is recommended to call with `noGarbage` set to `.TRUE.`, fully removing the entire temporary object from memory.

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

26.6.50 ESMF_FieldHaloStore - Store a FieldHalo operation

INTERFACE:

```
     subroutine ESMF_FieldHaloStore(field, routehandle,  &
       startregion, haloLDepth, haloUDepth, rc)
```

ARGUMENTS:

```
    type(ESMF_Field),             intent(inout)           :: field
    type(ESMF_RouteHandle),       intent(inout)           :: routehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_StartRegion_Flag), intent(in),    optional :: startregion
    integer,                     intent(in),    optional :: haloLDepth(:)
    integer,                     intent(in),    optional :: haloUDepth(:)
    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:

Store a Field halo operation over the data in `field`. By default, i.e. without specifying `startregion`, `haloLDepth` and `haloUDepth`, all elements in the total Field region that lie outside the exclusive region will be considered potential destination elements for halo. However, only those elements that have a corresponding halo source element, i.e. an exclusive element on one of the DEs, will be updated under the halo operation. Elements that have no associated source remain unchanged under halo.

Specifying `startregion` allows to change the shape of the effective halo region from the inside. Setting this flag to `ESMF_STARTREGION_COMPUTATIONAL` means that only elements outside the computational region of the Field are considered for potential destination elements for the halo operation. The default is `ESMF_STARTREGION_EXCLUSIVE`.

The `haloLDepth` and `haloUDepth` arguments allow to reduce the extent of the effective halo region. Starting at the region specified by `startregion`, the `haloLDepth` and `haloUDepth` define a halo depth in each direction. Note that the maximum halo region is limited by the total Field region, independent of the actual `haloLDepth` and `haloUDepth` setting. The total Field region is local DE specific. The `haloLDepth` and `haloUDepth` are interpreted as the maximum desired extent, reducing the potentially larger region available for the halo operation.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_FieldHalo()` on any Field that matches `field` in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This call is *collective* across the current VM.

field `ESMF_Field` containing data to be haloed. The data in this Field may be destroyed by this call.

routehandle Handle to the precomputed Route.

[startregion] The start of the effective halo region on every DE. The default setting is `ESMF_STARTREGION_EXCLUSIVE`, rendering all non-exclusive elements potential halo destination elements. See section **??** for a complete list of valid settings.

[haloLDepth] This vector specifies the lower corner of the effective halo region with respect to the lower corner of `startregion`. The size of `haloLDepth` must equal the number of distributed Array dimensions.

[haloUDepth] This vector specifies the upper corner of the effective halo region with respect to the upper corner of `startregion`. The size of `haloUDepth` must equal the number of distributed Array dimensions.

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

26.6.51 ESMF_FieldIsCreated - Check whether a Field object has been created

INTERFACE:

```
function ESMF_FieldIsCreated(field, rc)
```

RETURN VALUE:

```
logical :: ESMF_FieldIsCreated
```

ARGUMENTS:

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

DESCRIPTION:

Return `.true.` if the `field` has been created. Otherwise return `.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:

field `ESMF_Field` queried.

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

26.6.52 ESMF_FieldPrint - Print Field information

INTERFACE:

```
subroutine ESMF_FieldPrint(field, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in)            :: field
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
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:

Prints information about the `field` to `stdout`. This subroutine goes through the internal data members of a field data type and prints information of each data member.

The arguments are:

field An ESMF_Field object.

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

26.6.53 ESMF_FieldRead - Read Field data from a file

INTERFACE:

```
   subroutine ESMF_FieldRead(field, fileName,        &
       variableName, timeslice, iofmt, rc)
```

ARGUMENTS:

```
     type(ESMF_Field),      intent(inout)          :: field
     character(*),          intent(in)             :: fileName
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     character(*),          intent(in),  optional  :: variableName
     integer,               intent(in),  optional  :: timeslice
     type(ESMF_IOFmt_Flag), intent(in),  optional  :: iofmt
     integer,               intent(out), optional  :: rc
```

DESCRIPTION:

Read Field data from a file and put it into an ESMF_Field object. For this API to be functional, the environment variable ESMF_PIO should be set to "internal" when the ESMF library is built. Please see the section on Data I/O, **??**.

Limitations:

- Only single tile Fields are supported.
- Not supported in ESMF_COMM=mpiuni mode.

The arguments are:

field The ESMF_Field object in which the read data is returned.

fileName The name of the file from which Field data is read.

[variableName] Variable name in the file; default is the "name" of Field. Use this argument only in the I/O format (such as NetCDF) that supports variable name. If the I/O format does not support this (such as binary format), ESMF will return an error code.

timeslice Number of slices to be read from file, starting from the 1st slice

[iofmt] The I/O format. Please see Section **??** for the list of options. If not present, file names with a .bin extension will use ESMF_IOFMT_BIN, and file names with a .nc extension will use ESMF_IOFMT_NETCDF. Other files default to ESMF_IOFMT_NETCDF.

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

26.6.54 ESMF_FieldRedist - Execute a Field redistribution

INTERFACE:

```
   subroutine ESMF_FieldRedist(srcField, dstField, routehandle,  &
     checkflag, rc)
```

ARGUMENTS:

```
         type(ESMF_Field),       intent(in),optional     :: srcField
         type(ESMF_Field),       intent(inout),optional  :: dstField
         type(ESMF_RouteHandle), intent(inout)           :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
         logical,                intent(in),   optional  :: checkflag
         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:

Execute a precomputed Field redistribution from `srcField` to `dstField`. Both `srcField` and `dstField` must match the respective Fields used during `ESMF_FieldRedistStore()` in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

The `srcField` and `dstField` arguments are optional in support of the situation where `srcField` and/or `dstField` are not defined on all PETs. The `srcField` and `dstField` must be specified on those PETs that hold source or destination DEs, respectively, but may be omitted on all other PETs. PETs that hold neither source nor destination DEs may omit both arguments.

It is erroneous to specify the identical Field object for `srcField` and `dstField` arguments.

See `ESMF_FieldRedistStore()` on how to precompute `routehandle`.

This call is *collective* across the current VM.

For examples and associated documentation regarding this method see Section 26.3.30.

[srcField] `ESMF_Field` with source data.

[dstField] `ESMF_Field` with destination data.

routehandle Handle to the precomputed Route.

[checkflag] If set to `.TRUE.` the input Field pair will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

26.6.55 ESMF_FieldRedistRelease - Release resources associated with Field redistribution

INTERFACE:

```
   subroutine ESMF_FieldRedistRelease(routehandle, noGarbage, rc)
```

ARGUMENTS:

```
         type(ESMF_RouteHandle), intent(inout)           :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
         logical,                intent(in),   optional  :: noGarbage
         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:

  8.0.0 Added argument `noGarbage`. The argument provides a mechanism to override the default garbage collection mechanism when destroying an ESMF object.

DESCRIPTION:

Release resources associated with a Field redistribution. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

[noGarbage] If set to `.TRUE.` the object will be fully destroyed and removed from the ESMF garbage collection system. Note however that under this condition ESMF cannot protect against accessing the destroyed object through dangling aliases – a situation which may lead to hard to debug application crashes.

It is generally recommended to leave the `noGarbage` argument set to `.FALSE.` (the default), and to take advantage of the ESMF garbage collection system which will prevent problems with dangling aliases or incorrect sequences of destroy calls. However this level of support requires that a small remnant of the object is kept in memory past the destroy call. This can lead to an unexpected increase in memory consumption over the course of execution in applications that use temporary ESMF objects. For situations where the repeated creation and destruction of temporary objects leads to memory issues, it is recommended to call with `noGarbage` set to `.TRUE.`, fully removing the entire temporary object from memory.

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

26.6.56 ESMF_FieldRedistStore - Precompute Field redistribution with a local factor argument

INTERFACE:

```
! Private name; call using ESMF_FieldRedistStore()
subroutine ESMF_FieldRedistStore<type><kind>(srcField, dstField, &
       routehandle, factor, srcToDstTransposeMap, &
       ignoreUnmatchedIndices, rc)
```

ARGUMENTS:

```
     type(ESMF_Field),         intent(in)            :: srcField
     type(ESMF_Field),         intent(inout)         :: dstField
     type(ESMF_RouteHandle),   intent(inout)         :: routehandle
     <type>(ESMF_KIND_<kind>), intent(in)            :: factor
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,                  intent(in),  optional :: srcToDstTransposeMap(:)
     logical,                  intent(in),  optional :: ignoreUnmatchedIndices
     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 `ignoreUnmatchedIndices` to support sparse matrices that contain elements with indices that do not have a match within the source or destination Array.

DESCRIPTION:

`ESMF_FieldRedistStore()` is a collective method across all PETs of the current Component. The interface of the method is overloaded, allowing – in principle – each PET to call into `ESMF_FieldRedistStore()` through a different entry point. Restrictions apply as to which combinations are sensible. All other combinations result in ESMF run time errors. The complete semantics of the `ESMF_FieldRedistStore()` method, as provided through the separate entry points shown in 26.6.56 and 26.6.57, is described in the following paragraphs as a whole.

Store a Field redistribution operation from `srcField` to `dstField`. Interface 26.6.56 allows PETs to specify a `factor` argument. PETs not specifying a `factor` argument call into interface 26.6.57. If multiple PETs specify the `factor` argument, its type and kind, as well as its value must match across all PETs. If none of the PETs specify a `factor` argument the default will be a factor of 1. The resulting factor is applied to all of the source data during redistribution, allowing scaling of the data, e.g. for unit transformation.

Both `srcField` and `dstField` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 28.2.18 for details on the definition of *sequence indices*.

Source Field, destination Field, and the factor may be of different <type><kind>. Further, source and destination Fields may differ in shape, however, the number of elements must match.

If `srcToDstTransposeMap` is not specified the redistribution corresponds to an identity mapping of the sequentialized source Field to the sequentialized destination Field. If the `srcToDstTransposeMap` argument is provided it must be identical on all PETs. The `srcToDstTransposeMap` allows source and destination Field dimensions to be transposed during the redistribution. The number of source and destination Field dimensions must be equal under this condition and the size of mapped dimensions must match.

It is erroneous to specify the identical Field object for `srcField` and `dstField` arguments.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_FieldRedist()` on any pair of Fields that matches `srcField` and `dstField` in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This method is overloaded for:
`ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`,
`ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

This call is *collective* across the current VM.

For examples and associated documentation regarding this method see Section 26.3.30.

The arguments are:

srcField `ESMF_Field` with source data.

dstField `ESMF_Field` with destination data. The data in this Field may be destroyed by this call.

routehandle Handle to the precomputed Route.

factor Factor by which to multiply data. Default is 1. See full method description above for details on the interplay with other PETs.

[srcToDstTransposeMap] List with as many entries as there are dimensions in `srcField`. Each entry maps the corresponding `srcField` dimension against the specified `dstField` dimension. Mixing of distributed and undistributed dimensions is supported.

[ignoreUnmatchedIndices] A logical flag that affects the behavior for when not all elements match between the `srcField` and `dstField` side. The default setting is `.false.`, indicating that it is an error when such a situation is encountered. Setting `ignoreUnmatchedIndices` to `.true.` ignores unmatched indices.

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

26.6.57 ESMF_FieldRedistStore - Precompute Field redistribution without a local factor argument

INTERFACE:

```
! Private name; call using ESMF_FieldRedistStore()
  subroutine ESMF_FieldRedistStoreNF(srcField, dstField, &
      routehandle, srcToDstTransposeMap, &
      ignoreUnmatchedIndices, rc)
```

ARGUMENTS:

```
        type(ESMF_Field),       intent(in)            :: srcField
        type(ESMF_Field),       intent(inout)         :: dstField
        type(ESMF_RouteHandle), intent(inout)         :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
        integer,                intent(in), optional  :: srcToDstTransposeMap(:)
        logical,                intent(in), optional  :: ignoreUnmatchedIndices
        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:

`ESMF_FieldRedistStore()` is a collective method across all PETs of the current Component. The interface of the method is overloaded, allowing – in principle – each PET to call into `ESMF_FieldRedistStore()` through a different entry point. Restrictions apply as to which combinations are sensible. All other combinations result in ESMF run time errors. The complete semantics of the `ESMF_FieldRedistStore()` method, as provided through the separate entry points shown in 26.6.56 and 26.6.57, is described in the following paragraphs as a whole.

Store a Field redistribution operation from `srcField` to `dstField`. Interface 26.6.56 allows PETs to specify a `factor` argument. PETs not specifying a `factor` argument call into interface 26.6.57. If multiple PETs specify the `factor` argument, its type and kind, as well as its value must match across all PETs. If none of the PETs specify a `factor` argument the default will be a factor of 1. The resulting factor is applied to all of the source data during redistribution, allowing scaling of the data, e.g. for unit transformation.

Both `srcField` and `dstField` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 28.2.18 for details on the definition of *sequence indices*.

Source Field, destination Field, and the factor may be of different <type><kind>. Further, source and destination Fields may differ in shape, however, the number of elements must match.

If `srcToDstTransposeMap` is not specified the redistribution corresponds to an identity mapping of the sequentialized source Field to the sequentialized destination Field. If the `srcToDstTransposeMap` argument is provided it must be identical on all PETs. The `srcToDstTransposeMap` allows source and destination Field dimensions to be transposed during the redistribution. The number of source and destination Field dimensions must be equal under this condition and the size of mapped dimensions must match.

It is erroneous to specify the identical Field object for `srcField` and `dstField` arguments.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_FieldRedist()` on any pair of Fields that matches `srcField` and `dstField` in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This call is *collective* across the current VM.

For examples and associated documentation regarding this method see Section 26.3.30.

The arguments are:

srcField `ESMF_Field` with source data.

dstField `ESMF_Field` with destination data. The data in this Field may be destroyed by this call.

routehandle Handle to the precomputed Route.

[srcToDstTransposeMap] List with as many entries as there are dimensions in `srcField`. Each entry maps the corresponding `srcField` dimension against the specified `dstField` dimension. Mixing of distributed and undistributed dimensions is supported.

[ignoreUnmatchedIndices] A logical flag that affects the behavior for when not all elements match between the `srcField` and `dstField` side. The default setting is `.false.`, indicating that it is an error when such a situation is encountered. Setting `ignoreUnmatchedIndices` to `.true.` ignores unmatched indices.

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

26.6.58 ESMF_FieldRegrid - Compute a regridding operation

INTERFACE:

```
   subroutine ESMF_FieldRegrid(srcField, dstField, routehandle, &
     zeroregion, termorderflag, checkflag, dynamicMask, rc)
```

ARGUMENTS:

```
      type(ESMF_Field),               intent(in),    optional :: srcField
      type(ESMF_Field),               intent(inout), optional :: dstField
      type(ESMF_RouteHandle),         intent(inout)           :: routehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_Region_Flag),         intent(in),    optional :: zeroregion
      type(ESMF_TermOrder_Flag),      intent(in),    optional :: termorderflag
      logical,                        intent(in),    optional :: checkflag
      type(ESMF_DynamicMask), target, intent(in),    optional :: dynamicMask
      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:

  6.1.0 Added argument `termorderflag`. The new argument gives the user control over the order in which the src terms are summed up.

  7.1.0r Added argument `dynamicMask`. The new argument supports the dynamic masking feature.

DESCRIPTION:

Execute the precomputed regrid operation stored in `routehandle` to interpolate from `srcField` to `dstField`. See `ESMF_FieldRegridStore()` on how to precompute the `routehandle`.

Both srcField and dstField must match the respective Fields used during ESMF_FieldRegridStore() in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

The srcField and dstField arguments are optional in support of the situation where srcField and/or dstField are not defined on all PETs. The srcField and dstField must be specified on those PETs that hold source or destination DEs, respectively, but may be omitted on all other PETs. PETs that hold neither source nor destination DEs may omit both arguments.

It is erroneous to specify the identical Field object for srcField and dstField arguments.

This call is *collective* across the current VM.

[srcField] ESMF_Field with source data.

[dstField] ESMF_Field with destination data.

routehandle Handle to the precomputed Route.

[zeroregion] If set to ESMF_REGION_TOTAL *(default)* the total regions of all DEs in dstField will be initialized to zero before updating the elements with the results of the sparse matrix multiplication. If set to ESMF_REGION_EMPTY the elements in dstField will not be modified prior to the sparse matrix multiplication and results will be added to the incoming element values. Setting zeroregion to ESMF_REGION_SELECT will only zero out those elements in the destination Array that will be updated by the sparse matrix multiplication. See section **??** for a complete list of valid settings.

[termorderflag] Specifies the order of the source side terms in all of the destination sums. The termorderflag only affects the order of terms during the execution of the RouteHandle. See the **??** section for an in-depth discussion of *all* bit-for-bit reproducibility aspects related to route-based communication methods. See **??** for a full list of options. The default setting depends on whether the dynamicMask argument is present or not. With dynamicMask argument present, the default of termorderflag is ESMF_TERMORDER_SRCSEQ. This ensures that all source terms are present on the destination side, and the interpolation can be calculated as a single sum. When dynamicMask is absent, the default of termorderflag is ESMF_TERMORDER_FREE, allowing maximum flexibility and partial sums for optimum performance.

[checkflag] If set to .TRUE. the input Array pair will be checked for consistency with the precomputed operation provided by routehandle. If set to .FALSE. *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set checkflag to .FALSE. to achieve highest performance.

[dynamicMask] Object holding dynamic masking information. See section **??** for a discussion of dynamic masking.

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

26.6.59 ESMF_FieldRegridRelease - Free resources used by a regridding operation

INTERFACE:

```
subroutine ESMF_FieldRegridRelease(routehandle, &
  noGarbage, rc)
```

ARGUMENTS:

```
      type(ESMF_RouteHandle), intent(inout)         :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      logical,                intent(in),  optional :: noGarbage
      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:

  8.0.0 Added argument `noGarbage`. The argument provides a mechanism to override the default garbage collection mechanism when destroying an ESMF object.

DESCRIPTION:

Release resources associated with a regrid operation. After this call `routehandle` becomes invalid.

The arguments are:

routehandle Handle to the precomputed Route.

[noGarbage] If set to `.TRUE.` the object will be fully destroyed and removed from the ESMF garbage collection system. Note however that under this condition ESMF cannot protect against accessing the destroyed object through dangling aliases – a situation which may lead to hard to debug application crashes.

It is generally recommended to leave the `noGarbage` argument set to `.FALSE.` (the default), and to take advantage of the ESMF garbage collection system which will prevent problems with dangling aliases or incorrect sequences of destroy calls. However this level of support requires that a small remnant of the object is kept in memory past the destroy call. This can lead to an unexpected increase in memory consumption over the course of execution in applications that use temporary ESMF objects. For situations where the repeated creation and destruction of temporary objects leads to memory issues, it is recommended to call with `noGarbage` set to `.TRUE.`, fully removing the entire temporary object from memory.

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

26.6.60 ESMF_FieldRegridStore - Precompute a Field regridding operation and return a RouteHandle and weights

INTERFACE:

```
  !   Private name; call using ESMF_FieldRegridStore()
      subroutine ESMF_FieldRegridStoreNX(srcField, dstField, &
                   srcMaskValues, dstMaskValues, &
                   regridmethod, &
                   polemethod, regridPoleNPnts, &
```

```
                        lineType, &
                        normType, &
                        extrapMethod, &
                        extrapNumSrcPnts, &
                        extrapDistExponent, &
                        extrapNumLevels, &
                        unmappedaction, ignoreDegenerate, &
                        srcTermProcessing, &
                        pipeLineDepth, &
                        routehandle, &
                        factorList, factorIndexList, &
                        weights, indices, &  ! DEPRECATED ARGUMENTS
                        srcFracField, dstFracField, &
                        dstStatusField, &
                        unmappedDstList, &
                        checkFlag, &
                        rc)
```

ARGUMENTS:

```
      type(ESMF_Field),               intent(in)             :: srcField
      type(ESMF_Field),               intent(inout)          :: dstField
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
      integer(ESMF_KIND_I4),          intent(in),    optional :: srcMaskValues(:)
      integer(ESMF_KIND_I4),          intent(in),    optional :: dstMaskValues(:)
      type(ESMF_RegridMethod_Flag),   intent(in),    optional :: regridmethod
      type(ESMF_PoleMethod_Flag),     intent(in),    optional :: polemethod
      integer,                        intent(in),    optional :: regridPoleNPnts
      type(ESMF_LineType_Flag),       intent(in),    optional :: lineType
      type(ESMF_NormType_Flag),       intent(in),    optional :: normType
      type(ESMF_ExtrapMethod_Flag),   intent(in),    optional :: extrapMethod
      integer,                        intent(in),    optional :: extrapNumSrcPnts
      real(ESMF_KIND_R4),             intent(in),    optional :: extrapDistExponent
      integer,                        intent(in),    optional :: extrapNumLevels
      type(ESMF_UnmappedAction_Flag), intent(in),    optional :: unmappedaction
      logical,                        intent(in),    optional :: ignoreDegenerate
      integer,                        intent(inout), optional :: srcTermProcessing
      integer,                        intent(inout), optional :: pipeLineDepth
      type(ESMF_RouteHandle),         intent(inout), optional :: routehandle
      real(ESMF_KIND_R8),             pointer,       optional :: factorList(:)
      integer(ESMF_KIND_I4),          pointer,       optional :: factorIndexList(:,:)
      real(ESMF_KIND_R8),    pointer, optional :: weights(:)   ! DEPRECATED ARG
      integer(ESMF_KIND_I4), pointer, optional :: indices(:,:) ! DEPRECATED ARG
      type(ESMF_Field),               intent(inout), optional :: srcFracField
      type(ESMF_Field),               intent(inout), optional :: dstFracField
      type(ESMF_Field),               intent(inout), optional :: dstStatusField
      integer(ESMF_KIND_I4),          pointer,       optional :: unmappedDstList(:)
      logical,                        intent(in),    optional :: checkFlag
      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:

  5.2.0rp1 Added arguments `factorList` and `factorIndexList`. Started to deprecate arguments `weights` and `indices`. This corrects an inconsistency of this interface with all other ESMF methods that take these same arguments.

  6.1.0 Added arguments `ignoreDegenerate`, `srcTermProcessing`, `pipelineDepth`, and `unmappedDstList`. The argument `ignoreDegenerate` allows the user to skip degenerate cells in the regridding instead of stopping with an error. The two arguments `srcTermProcessing` and `pipelineDepth` provide access to the tuning parameters affecting the sparse matrix execution. The argument `unmappedDstList` allows the user to get a list of the destination items which the regridding couldn't map to a source.

  6.3.0r Added argument `lineType`. This argument allows the user to control the path of the line between two points on a sphere surface. This allows the user to use their preferred line path for the calculation of distances and the shape of cells during regrid weight calculation on a sphere.

  6.3.0rp1 Added argument `normType`. This argument allows the user to control the type of normalization done during conservative weight generation.

  7.1.0r Added argument `dstStatusField`. This argument allows the user to receive information about what happened to each location in the destination Field during regridding.

  Added arguments `extrapMethod`, `extrapNumSrcPnts`, and `extrapDistExponent`. These three new extrapolation arguments allow the user to extrapolate destination points not mapped by the regrid method. `extrapMethod` allows the user to choose the extrapolation method. `extrapNumSrcPnts` and `extrapDistExponent` are parameters that allow the user to tune the behavior of the `ESMF_EXTRAPMETHOD_NEAREST_IDAVG` method.

  8.0.0 Added argument `extrapNumLevels`. For level based extrapolation methods (e.g. `ESMF_EXTRAPMETHOD_CREEP`) this argument allows the user to set how many levels to extrapolate.

  8.1.0 Added argument `checkFlag` to enable the user to turn on more expensive error checking during regrid weight calculation.

DESCRIPTION:

Creates a sparse matrix operation (stored in `routehandle`) that contains the calculations and communications necessary to interpolate from `srcField` to `dstField`. The routehandle can then be used in the call `ESMF_FieldRegrid()` to interpolate between the Fields. The user may also get the interpolation matrix in sparse matrix form via the optional arguments `factorList` and `factorIndexList`.

The routehandle generated by this call is based just on the coordinates in the spatial class (e.g. Grid) contained in the Fields. If those coordinates don't change the routehandle can be used repeatedly to interpolate from the source Field to the destination Field. This is true even if the data in the Fields changes. The routehandle may also be used to interpolate between any source and destination Field which are created on the same location in the same Grid, LocStream, XGrid, or Mesh as the original Fields.

When it's no longer needed the routehandle should be destroyed by using `ESMF_FieldRegridRelease()` to free the memory it's using.

Note, as a side effect, that this call may change the data in `dstField`. If this is undesirable, then an easy work around is to create a second temporary field with the same structure as `dstField` and pass that in instead.

The arguments are:

srcField Source Field.

dstField Destination Field. The data in this Field may be overwritten by this call.

[srcMaskValues] Mask information can be set in the Grid (see 31.3.17) or Mesh (see **??**) upon which the `srcField` is built. The `srcMaskValues` argument specifies the values in that mask information which indicate a source point should be masked out. In other words, a locati on is masked if and only if the value for that location in the mask information matches one of the values listed in `srcMaskValues`. If `srcMaskValues` is not specified, no masking will occur.

[dstMaskValues] Mask information can be set in the Grid (see 31.3.17) or Mesh (see **??**) upon which the `dstField` is built. The `dstMaskValues` argument specifies the values in that mask information which indicate a destination point should be masked out. In other words, a location is masked if and only if the value for that location in the mask information matches one of the values listed in `dstMaskValues`. If `dstMaskValues` is not specified, no masking will occur.

[regridmethod] The type of interpolation. Please see Section **??** for a list of valid options. If not specified, defaults to `ESMF_REGRIDMETHOD_BILINEAR`.

[polemethod] Specifies the type of pole to construct on the source Grid during regridding. Please see Section **??** for a list of valid options. If not specified, defaults to `ESMF_POLEMETHOD_ALLAVG` for non-conservative regrid methods, and `ESMF_POLEMETHOD_NONE` for conservative methods.

[regridPoleNPnts] If `polemethod` is `ESMF_POLEMETHOD_NPNTAVG`, then this parameter indicates the number of points over which to average. If `polemethod` is not ESMF_POLEMETHOD_NPNTAVG and `regridPoleNPnts` is specified, then it will be ignored. This subroutine will return an error if `polemethod` is ESMF_POLEMETHOD_NPNTAVG and `regridPoleNPnts` is not specified.

[lineType] This argument controls 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. As would be expected, this argument is only applicable when `srcField` and `dstField` are built on grids which lie on the surface of a sphere. Section **??** shows a list of valid options for this argument. If not specified, the default depends on the regrid method. Section **??** has the defaults by line type. Figure 24.2.16 shows which line types are supported for each regrid method as well as showing the default line type by regrid method.

[normType] This argument controls the type of normalization used when generating conservative weights. This option only applies to weights generated with `regridmethod=ESMF_REGRIDMETHOD_CONSERVE` or `regridmethod=ESMF_REGRIDMETHOD_CONSERVE_2ND` Please see Section **??** for a list of valid options. If not specified `normType` defaults to `ESMF_NORMTYPE_DSTAREA`.

[extrapMethod] The type of extrapolation. Please see Section **??** for a list of valid options. If not specified, defaults to `ESMF_EXTRAPMETHOD_NONE`.

[extrapNumSrcPnts] The number of source points to use for the extrapolation methods that use more than one source point (e.g. `ESMF_EXTRAPMETHOD_NEAREST_IDAVG`). If not specified, defaults to 8.

[extrapDistExponent] The exponent to raise the distance to when calculating weights for the `ESMF_EXTRAPMETHOD_NEAREST_IDAVG` extrapolation method. A higher value reduces the influence of more distant points. If not specified, defaults to 2.0.

[extrapNumLevels] The number of levels to output for the extrapolation methods that fill levels (e.g. `ESMF_EXTRAPMETHOD_CREEP`). When a method is used that requires this, then an error will be returned, if it is not specified.

[unmappedaction] Specifies what should happen if there are destination points that can't be mapped to a source cell. Please see Section **??** for a list of valid options. If not specified, `unmappedaction` defaults to `ESMF_UNMAPPEDACTION_ERROR`.

[ignoreDegenerate] Ignore degenerate cells when checking for errors. If this is set to true, then the regridding proceeds, but degenerate cells will be skipped. If set to false, a degenerate cell produces an error. If not specified, `ignoreDegenerate` defaults to false.

[srcTermProcessing] The `srcTermProcessing` parameter controls how many source terms, located on the same PET and summing into the same destination element, are summed into partial sums on the source PET before being transferred to the destination PET. A value of 0 indicates that the entire arithmetic is done on the destination PET; source elements are neither multiplied by their factors nor added into partial sums before being sent off by the source PET. A value of 1 indicates that source elements are multiplied by their factors on the source side before being sent to the destination PET. Larger values of `srcTermProcessing` indicate the maximum number of terms in the partial sums on the source side.

Note that partial sums may lead to bit-for-bit differences in the results. See section **??** for an in-depth discussion of *all* bit-for-bit reproducibility aspects related to route-based communication methods.

The `ESMF_FieldRegridStore()` method implements an auto-tuning scheme for the `srcTermProcessing` parameter. The intent on the `srcTermProcessing` argument is `"inout"` in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `srcTermProcessing` parameter, and the auto-tuning phase is skipped. In this case the `srcTermProcessing` argument is not modified on return. If the provided argument is < 0, the `srcTermProcessing` parameter is determined internally using the auto-tuning scheme. In this case the `srcTermProcessing` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `srcTermProcessing` argument is omitted.

[pipelineDepth] The `pipelineDepth` parameter controls how many messages a PET may have outstanding during a sparse matrix exchange. Larger values of `pipelineDepth` typically lead to better performance. However, on some systems too large a value may lead to performance degradation, or runtime errors.

Note that the pipeline depth has no effect on the bit-for-bit reproducibility of the results. However, it may affect the performance reproducibility of the exchange.

The `ESMF_FieldRegridStore()` method implements an auto-tuning scheme for the `pipelineDepth` parameter. The intent on the `pipelineDepth` argument is `"inout"` in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `pipelineDepth` parameter, and the auto-tuning phase is skipped. In this case the `pipelineDepth` argument is not modified on return. If the provided argument is < 0, the `pipelineDepth` parameter is determined internally using the auto-tuning scheme. In this case the `pipelineDepth` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `pipelineDepth` argument is omitted.

[routehandle] The communication handle that implements the regrid operation and that can be used later in the `ESMF_FieldRegrid()` call. The `routehandle` is optional so that if the user doesn't need it, then they can indicate that by not requesting it. The time to compute the `routehandle` can be a significant fraction of the time taken by this method, so if it's not needed then not requesting it is worthwhile.

[factorList] The list of coefficients for a sparse matrix which interpolates from `srcField` to `dstField`. The array coming out of this variable is in the appropriate format to be used in other ESMF sparse matrix multiply calls, for example `ESMF_FieldSMMStore()`. The `factorList` array is allocated by the method and the user is responsible for deallocating it.

[factorIndexList] The indices for a sparse matrix which interpolates from `srcField` to `dstField`. This argument is a 2D array containing pairs of source and destination sequence indices corresponding to the coefficients in the `factorList` argument. The first dimension of `factorIndexList` is of size 2. `factorIndexList(1,:)` specifies the sequence index of the source element in the `srcField`. `factorIndexList(2,:)` specifies the sequence index of the destination element in the `dstField`. The se cond dimension of `factorIndexList` steps through the list of pairs, i.e. `size(factorIndexList,2)==size(factorList)`. The array coming out of this variable is in the appropriate format to be used in other ESMF sparse matrix multiply calls, for example `ESMF_FieldSMMStore()`. The `factorIndexList` array is allocated by the method and the user is responsible for deallocating it.

[weights] DEPRECATED ARGUMENT! Please use the argument `factorList` instead.

[indices] DEPRECATED ARGUMENT! Please use the argument `factorIndexList` instead.

[srcFracField] The fraction of each source cell participating in the regridding. Only valid when regridmethod is ESMF_REGRIDMETHOD_CONSERVE or regridmethod=ESMF_REGRIDMETHOD_CONSERVE_2ND. This Field needs to be created on the same location (e.g staggerloc) as the srcField.

[dstFracField] The fraction of each destination cell participating in the regridding. Only valid when regridmethod is ESMF_REGRIDMETHOD_CONSERVE or regridmethod=ESMF_REGRIDMETHOD_CONSERVE_2ND. This Field needs to be created on the same location (e.g staggerloc) as the dstField. It is important to note that the current implementation of 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.

[dstStatusField] An ESMF Field which outputs a regrid status value for each destination location. Section **??** indicates the meaning of each value. The Field needs to be built on the same location (e.g. staggerloc) in the same Grid, Mesh, XGrid, or LocStream as the dstField argument. The Field also needs to be of typekind ESMF_TYPEKIND_I4. This option currently doesn't work with the ESMF_REGRIDMETHOD_NEAREST_DTOS regrid method.

[unmappedDstList] The list of the sequence indices for locations in dstField which couldn't be mapped the srcField. The list on each PET only contains the unmapped locations for the piece of the dstField on that PET. If a destination point is masked, it won't be put in this list. This option currently doesn't work with the ESMF_REGRIDMETHOD_NEAREST_DTOS regrid method.

[checkFlag] If set to .FALSE. *(default)* only quick error checking will be performed. If set to .TRUE. more expensive error checking will be performed, possibly catching more errors. Set checkFlag to .FALSE. to achieve highest performance. The checkFlag currently only turns on checking for conservative regrid methods (e.g. ESMF_REGRIDMETHOD_CONSERVE).

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

26.6.61 ESMF_FieldRegridStore - Precompute a Field regridding operation between an XGrid and one of its side Grids or Meshes

INTERFACE:

```
   !   Private name; call using ESMF_FieldRegridStore()
       subroutine ESMF_FieldRegridStoreX(xgrid, srcField, dstField, &
                    regridmethod, routehandle, &
                    srcFracField, dstFracField, &
                    srcMergeFracField, dstMergeFracField, rc)
```

ARGUMENTS:

```
       type(ESMF_XGrid),       intent(in)              :: xgrid
       type(ESMF_Field),       intent(in)              :: srcField
       type(ESMF_Field),       intent(inout)           :: dstField
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       type(ESMF_RegridMethod_Flag),   intent(in),    optional :: regridmethod
       type(ESMF_RouteHandle), intent(inout), optional :: routehandle
       type(ESMF_Field),       intent(inout), optional :: srcFracField
```

```
type(ESMF_Field),       intent(inout), optional :: dstFracField
type(ESMF_Field),       intent(inout), optional :: srcMergeFracField
type(ESMF_Field),       intent(inout), optional :: dstMergeFracField
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:

  5.3.0 Added arguments `srcFracField`, `dstFracField`, `srcMergeFracField`, and `dstMergeFracField`. These fraction Fields allow a user to calculate correct flux regridded through `ESMF_XGrid`.

  7.1.0r Added argument `regridmethod`. This new argument allows the user to choose the regrid method to apply when computing the routehandle.

DESCRIPTION:

This method creates a RouteHandle to do conservative interpolation specifically between a Field built on an XGrid and a Field build on one of the Grids or Meshes used to create that XGrid. (To do more general interpolation use the `ESMF_FieldRegridStore()` method in section 26.6.60.) The RouteHandle produced by this method can then be used in the call `ESMF_FieldRegrid()` to interpolate from the `srcField` to the `dstField`.

The RouteHandle generated by this call is based just on the coordinates in the Grids, XGrids, or Meshes contained in the Fields. If those coordinates don't change the RouteHandle can be used repeatedly to interpolate from the source Field to the destination Field. This is true even if the data in the Fields changes. The RouteHandle may also be used to interpolate between any source and destination Field which are created on the same Grid, XGrid, or Mesh as the original Fields.

When it's no longer needed the RouteHandle should be destroyed by using `ESMF_FieldRegridRelease()` to free the memory it's using.

Note, as a side effect, that this call may change the data in `dstField`. If this is undesirable, then an easy work around is to create a second temporary Field with the same structure as `dstField` and pass that in instead.

The arguments are:

xgrid Exchange Grid.

srcField Source Field built on either `xgrid` or one of the Grids or Meshes used to create `xgrid`.

dstField Destination Field built on either `xgrid` or one of the Grids or Meshes used to create `xgrid`. The data in this Field may be overwritten by this call.

[regridmethod] The type of interpolation. For this method only `ESMF_REGRIDMETHOD_CONSERVE` and `ESMF_REGRIDMETHOD_CONSERVE_2ND` are supported. If not specified, defaults to `ESMF_REGRIDMETHOD_CONSERVE`.

[routehandle] The handle that implements the regrid and that can be used in later `ESMF_FieldRegrid`.

[srcFracField] The fraction of each source cell participating in the regridding returned from this call. This Field needs to be created on the same Grid and location (e.g staggerloc) as the srcField.

[dstFracField] The fraction of each destination cell participating in the regridding returned from this call. This Field needs to be created on the same Grid and location (e.g staggerloc) as the dstField.

[srcMergeFracField] The fraction of each source cell as a result of Grid merge returned from this call. This Field needs to be created on the same Grid and location (e.g staggerloc) as the srcField.

[dstMergeFracField] The fraction of each destination cell as a result of Grid merge returned from this call. This Field needs to be created on the same Grid and location (e.g staggerloc) as the dstField.

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

26.6.62 ESMF_FieldRegridGetArea - Get the area of the cells used for conservative interpolation

INTERFACE:

```
subroutine ESMF_FieldRegridGetArea(areaField, rc)
```

RETURN VALUE:

ARGUMENTS:

```
type(ESMF_Field), intent(inout)                 :: areaField
integer, intent(out), optional                  :: rc
```

DESCRIPTION:

This subroutine gets the area of the cells used for conservative interpolation for the grid object associated with `areaField` and puts them into `areaField`. If created on a 2D Grid, it must be built on the `ESMF_STAGGERLOC_CENTER` stagger location. If created on a 3D Grid, it must be built on the `ESMF_STAGGERLOC_CENTER_VCENTER` stagger location. If created on a Mesh, it must be built on the `ESMF_MESHLOC_ELEMENT` mesh location.

If the user has set the area in the Grid or Mesh under `areaField`, then that's the area that's returned in the units that the user set it in. If the user hasn't set the area, then the area is calculated and returned. If the Grid or Mesh is on the surface of a sphere, then the calculated area is in units of square radians. If the Grid or Mesh is Cartesian, then the calculated area is in square units of whatever unit the coordinates are in.

The arguments are:

areaField The Field to put the area values in.

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

26.6.63 ESMF_FieldScatter - Scatter a Fortran array across the ESMF_Field

INTERFACE:

```
subroutine ESMF_FieldScatter<rank><type><kind>(field, farray, &
rootPet, tile, vm, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(inout) :: field
mtype (ESMF_KIND_mtypekind),intent(in), target :: farray(mdim)
integer, intent(in) :: rootPet
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
integer, intent(in), optional :: tile
type(ESMF_VM), intent(in), 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.

DESCRIPTION:

Scatter the data of `farray` located on `rootPET` across an ESMF_Field object. A single `farray` must be scattered across a single DistGrid tile in Field. The optional `tile` argument allows selection of the tile. For Fields defined on a single tile DistGrid the default selection (tile 1) will be correct. The shape of `farray` must match the shape of the tile in Field.

If the Field contains replicating DistGrid dimensions data will be scattered across all of the replicated pieces.

The implementation of Scatter and Gather is not sequence index based. If the Field is built on arbitrarily distributed Grid, Mesh, LocStream or XGrid, Scatter will not scatter data from rootPet to the destination data points corresponding to the sequence index on the rootPet. Instead Scatter will scatter a contiguous memory range from rootPet to destination PET. The size of the memory range is equal to the number of data elements on the destination PET. Vice versa for the Gather operation. In this case, the user should use `ESMF_FieldRedist` to achieve the same data operation result. For examples how to use `ESMF_FieldRedist` to perform Gather and Scatter, please refer to 26.3.32 and 26.3.31.

This version of the interface implements the PET-based blocking paradigm: Each PET of the VM must issue this call exactly once for *all* of its DEs. The call will block until all PET-local data objects are accessible.

For examples and associated documentation regarding this method see Section 26.3.29.

The arguments are:

field The `ESMF_Field` object across which data will be scattered.

{farray} The Fortran array that is to be scattered. Only root must provide a valid `farray`, the other PETs may treat `farray` as an optional argument.

rootPet PET that holds the valid data in `farray`.

[tile] The DistGrid tile in `field` into which to scatter `farray`. By default `farray` will be scattered into tile 1.

[vm] Optional `ESMF_VM` object of the current context. Providing the VM of the current context will lower the method's overhead.

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

26.6.64 ESMF_FieldSet - Set object-wide Field information

INTERFACE:

```
   subroutine ESMF_FieldSet(field, name, rc)
```

ARGUMENTS:

```
     type(ESMF_Field),   intent(inout)         :: field
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     character(len = *), intent(in),  optional :: name
     integer,            intent(out), optional :: rc
```

DESCRIPTION:

Sets adjustable settings in an `ESMF_Field` object.

The arguments are:

field `ESMF_Field` object for which to set properties.

[name] The Field name.

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

26.6.65 ESMF_FieldSync - Synchronize DEs across the Field in case of sharing

INTERFACE:

```
   subroutine ESMF_FieldSync(field, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Synchronizes access to DEs across `field` to make sure PETs correctly access the data for read and write when DEs are shared.

The arguments are:

field Specified `ESMF_Field` object.

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

26.6.66 ESMF_FieldSMM - Execute a Field sparse matrix multiplication

INTERFACE:

```
   subroutine ESMF_FieldSMM(srcField, dstField, routehandle, &
              zeroregion, termorderflag, checkflag, rc)
```

ARGUMENTS:

```
         type(ESMF_Field),          intent(in),    optional  :: srcField
         type(ESMF_Field),          intent(inout), optional  :: dstField
         type(ESMF_RouteHandle),    intent(inout)            :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
         type(ESMF_Region_Flag),    intent(in),    optional  :: zeroregion
         type(ESMF_TermOrder_Flag), intent(in),    optional  :: termorderflag
         logical,                   intent(in),    optional  :: checkflag
         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.

  6.1.0 Added argument `termorderflag`. The new argument gives the user control over the order in which the src terms are summed up.

DESCRIPTION:

Execute a precomputed Field sparse matrix multiplication from `srcField` to `dstField`. Both `srcField` and `dstField` must match the respective Fields used during `ESMF_FieldSMMStore()` in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

The `srcField` and `dstField` arguments are optional in support of the situation where `srcField` and/or `dstField` are not defined on all PETs. The `srcField` and `dstField` must be specified on those PETs that hold source or destination DEs, respectively, but may be omitted on all other PETs. PETs that hold neither source nor destination DEs may omit both arguments.

It is erroneous to specify the identical Field object for `srcField` and `dstField` arguments.

See `ESMF_FieldSMMStore()` on how to precompute `routehandle`.

This call is *collective* across the current VM.

For examples and associated documentation regarding this method see Section 26.3.33.

[srcField] `ESMF_Field` with source data.

[dstField] `ESMF_Field` with destination data.

routehandle Handle to the precomputed Route.

[zeroregion] If set to `ESMF_REGION_TOTAL` *(default)* the total regions of all DEs in `dstField` will be initialized to zero before updating the elements with the results of the sparse matrix multiplication. If set to `ESMF_REGION_EMPTY` the elements in `dstField` will not be modified prior to the sparse matrix multiplication and results will be added to the incoming element values. Setting `zeroregion` to `ESMF_REGION_SELECT` will only zero out those elements in the destination Field that will be updated by the sparse matrix multiplication. See section **??** for a complete list of valid settings.

[termorderflag] Specifies the order of the source side terms in all of the destination sums. The `termorderflag` only affects the order of terms during the execution of the RouteHandle. See the **??** section for an in-depth discussion of *all* bit-for-bit reproducibility aspects related to route-based communication methods. See **??** for a full list of options. The default is `ESMF_TERMORDER_FREE`, allowing maximum flexibility in the order of terms for optimum performance.

[checkflag] If set to `.TRUE.` the input Field pair will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

26.6.67 ESMF_FieldSMMRelease - Release resources associated with Field

sparse matrix multiplication

INTERFACE:

```
   subroutine ESMF_FieldSMMRelease(routehandle, noGarbage, rc)
```

ARGUMENTS:

```
         type(ESMF_RouteHandle), intent(inout)           :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
         logical,                intent(in),   optional  :: noGarbage
         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:

8.0.0 Added argument `noGarbage`. The argument provides a mechanism to override the default garbage collection mechanism when destroying an ESMF object.

DESCRIPTION:

Release resources associated with a Field sparse matrix multiplication. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

[noGarbage] If set to `.TRUE.` the object will be fully destroyed and removed from the ESMF garbage collection system. Note however that under this condition ESMF cannot protect against accessing the destroyed object through dangling aliases – a situation which may lead to hard to debug application crashes.

It is generally recommended to leave the `noGarbage` argument set to `.FALSE.` (the default), and to take advantage of the ESMF garbage collection system which will prevent problems with dangling aliases or incorrect sequences of destroy calls. However this level of support requires that a small remnant of the object is kept in memory past the destroy call. This can lead to an unexpected increase in memory consumption over the course of execution in applications that use temporary ESMF objects. For situations where the repeated creation and destruction of temporary objects leads to memory issues, it is recommended to call with `noGarbage` set to `.TRUE.`, fully removing the entire temporary object from memory.

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

26.6.68 ESMF_FieldSMMStore - Precompute Field sparse matrix multiplication with local factors

INTERFACE:

```
! Private name; call using ESMF_FieldSMMStore()
subroutine ESMF_FieldSMMStore<type><kind>(srcField, dstField, &
       routehandle, factorList, factorIndexList, &
       ignoreUnmatchedIndices, srcTermProcessing, pipelineDepth, rc)
```

ARGUMENTS:

```
     type(ESMF_Field),         intent(in)              :: srcField
     type(ESMF_Field),         intent(inout)           :: dstField
     type(ESMF_RouteHandle),   intent(inout)           :: routehandle
     <type>(ESMF_KIND_<kind>), intent(in)              :: factorList(:)
     integer,                  intent(in),             :: factorIndexList(:,:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                  intent(in),    optional :: ignoreUnmatchedIndices
     integer,                  intent(inout), optional :: srcTermProcessing
     integer,                  intent(inout), optional :: pipeLineDepth
     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:

  6.1.0 Added arguments `srcTermProcessing`, `pipelineDepth` The two arguments `srcTermProcessing` and `pipelineDepth` provide access to the tuning parameters affecting the sparse matrix execution.

  7.0.0 Added argument `transposeRoutehandle` to allow a handle to the transposed matrix operation to be returned.
  Added argument `ignoreUnmatchedIndices` to support sparse matrices that contain elements with indices that do not have a match within the source or destination Array.

  7.1.0r Removed argument `transposeRoutehandle` and provide it via interface overloading instead. This allows argument `srcField` to stay strictly intent(in) for this entry point.

DESCRIPTION:

Store a Field sparse matrix multiplication operation from `srcField` to `dstField`. PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* `factorList` and `factorIndexList` arguments.

Both `srcField` and `dstField` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 28.2.18 for details on the definition of *sequence indices*. SMM corresponds to an identity mapping of the source Field vector to the destination Field vector.

Source and destination Fields may be of different <type><kind>. Further source and destination Fields may differ in shape, however, the number of elements must match.

It is erroneous to specify the identical Field object for srcField and dstField arguments.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_FieldSMM()` on any pair of Fields that matches `srcField` and `dstField` in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This method is overloaded for:
`ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`,
`ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

This call is collective across the current VM.

For examples and associated documentation regarding this method see Section 26.3.33.

The arguments are:

srcField `ESMF_Field` with source data.

dstField `ESMF_Field` with destination data. The data in this Field may be destroyed by this call.

routehandle Handle to the precomputed Route.

factorList List of non-zero coefficients.

factorIndexList Pairs of sequence indices for the factors stored in `factorList`.

The second dimension of `factorIndexList` steps through the list of pairs, i.e. `size(factorIndexList,2) == size(factorList)`. The first dimension of `factorIndexList` is either of size 2 or size 4.

The second dimension of `factorIndexList` steps through the list of

In the *size 2 format* `factorIndexList(1,:)` specifies the sequence index of the source element in the `srcField` while `factorIndexList(2,:)` specifies the sequence index of the destination element in `dstField`. For this format to be a valid option source and destination Fields must have matching number of tensor elements (the product of the sizes of all Field tensor dimensions). Under this condition an identity matrix can be applied within the space of tensor elements for each sparse matrix factor.

The *size 4 format* is more general and does not require a matching tensor element count. Here the `factorIndexList(1,:)` specifies the sequence index while `factorIndexList(2,:)` specifies the tensor sequence index of the source element in the `srcField`. Further `factorIndexList(3,:)` specifies the sequence index and `factorIndexList(4,:)` specifies the tensor sequence index of the destination element in the `dstField`.

See section 28.2.18 for details on the definition of Field *sequence indices* and *tensor sequence indices*.

[ignoreUnmatchedIndices] A logical flag that affects the behavior for when sequence indices in the sparse matrix are encountered that do not have a match on the `srcField` or `dstField` side. The default setting is `.false.`, indicating that it is an error when such a situation is encountered. Setting `ignoreUnmatchedIndices` to `.true.` ignores entries with unmatched indices.

[srcTermProcessing] The `srcTermProcessing` parameter controls how many source terms, located on the same PET and summing into the same destination element, are summed into partial sums on the source PET before being transferred to the destination PET. A value of 0 indicates that the entire arithmetic is done on the destination PET; source elements are neither multiplied by their factors nor added into partial sums before being sent off by the source PET. A value of 1 indicates that source elements are multiplied by their factors on the source side before being sent to the destination PET. Larger values of `srcTermProcessing` indicate the maximum number of terms in the partial sums on the source side.

Note that partial sums may lead to bit-for-bit differences in the results. See section **??** for an in-depth discussion of *all* bit-for-bit reproducibility aspects related to route-based communication methods.

The `ESMF_FieldSMMStore()` method implements an auto-tuning scheme for the `srcTermProcessing` parameter. The intent on the `srcTermProcessing` argument is `"inout"` in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `srcTermProcessing` parameter, and the auto-tuning phase is skipped. In this case the `srcTermProcessing` argument is not modified on return. If the provided argument is < 0, the `srcTermProcessing` parameter is determined internally using the auto-tuning scheme. In this case the `srcTermProcessing` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `srcTermProcessing` argument is omitted.

[pipelineDepth] The `pipelineDepth` parameter controls how many messages a PET may have outstanding during a sparse matrix exchange. Larger values of `pipelineDepth` typically lead to better performance. However, on some systems too large a value may lead to performance degradation, or runtime errors.

Note that the pipeline depth has no effect on the bit-for-bit reproducibility of the results. However, it may affect the performance reproducibility of the exchange.

The `ESMF_FieldSMMStore()` method implements an auto-tuning scheme for the `pipelineDepth` parameter. The intent on the `pipelineDepth` argument is `"inout"` in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `pipelineDepth` parameter, and the auto-tuning phase is skipped. In this case the `pipelineDepth` argument is not modified on return. If the provided argument is < 0, the `pipelineDepth` parameter is determined internally using the

auto-tuning scheme. In this case the `pipelineDepth` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `pipelineDepth` argument is omitted.

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

26.6.69 ESMF_FieldSMMStore - Precompute Field sparse matrix multiplication and transpose with local factors

INTERFACE:

```
! Private name; call using ESMF_FieldSMMStore()
subroutine ESMF_FieldSMMStore<type><kind>TR(srcField, dstField, &
       routehandle, transposeRoutehandle, factorList, factorIndexList, &
       ignoreUnmatchedIndices, srcTermProcessing, &
       pipelineDepth, rc)
```

ARGUMENTS:

```
  type(ESMF_Field),          intent(inout)           :: srcField
  type(ESMF_Field),          intent(inout)           :: dstField
  type(ESMF_RouteHandle),    intent(inout)           :: routehandle
  type(ESMF_RouteHandle),    intent(inout)           :: transposeRoutehandle
  <type>(ESMF_KIND_<kind>), intent(in)              :: factorList(:)
  integer,                   intent(in),             :: factorIndexList(:,:)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  logical,                   intent(in),    optional :: ignoreUnmatchedIndices
  integer,                   intent(inout), optional :: srcTermProcessing
  integer,                   intent(inout), optional :: pipeLineDepth
  integer,                   intent(out),   optional :: rc
```

DESCRIPTION:

Store a Field sparse matrix multiplication operation from `srcField` to `dstField`. PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* `factorList` and `factorIndexList` arguments.

Both `srcField` and `dstField` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 28.2.18 for details on the definition of *sequence indices*. SMM corresponds to an identity mapping of the source Field vector to the destination Field vector.

Source and destination Fields may be of different <type><kind>. Further source and destination Fields may differ in shape, however, the number of elements must match.

It is erroneous to specify the identical Field object for srcField and dstField arguments.

The routine returns an ESMF_RouteHandle that can be used to call ESMF_FieldSMM() on any pair of Fields that matches srcField and dstField in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This method is overloaded for:
ESMF_TYPEKIND_I4, ESMF_TYPEKIND_I8,
ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8.

This call is collective across the current VM.

For examples and associated documentation regarding this method see Section 26.3.33.

The arguments are:

srcField ESMF_Field with source data. The data in this Array may be destroyed by this call.

dstField ESMF_Field with destination data. The data in this Field may be destroyed by this call.

routehandle Handle to the precomputed Route.

transposeRoutehandle A handle to the transposed matrix operation is returned. The transposed operation goes from dstArray to srcArray.

factorList List of non-zero coefficients.

factorIndexList Pairs of sequence indices for the factors stored in factorList.

The second dimension of factorIndexList steps through the list of pairs, i.e. size(factorIndexList,2) == size(factorList). The first dimension of factorIndexList is either of size 2 or size 4.

The second dimension of factorIndexList steps through the list of

In the *size 2 format* factorIndexList(1,:) specifies the sequence index of the source element in the srcField while factorIndexList(2,:) specifies the sequence index of the destination element in dstField. For this format to be a valid option source and destination Fields must have matching number of tensor elements (the product of the sizes of all Field tensor dimensions). Under this condition an identity matrix can be applied within the space of tensor elements for each sparse matrix factor.

The *size 4 format* is more general and does not require a matching tensor element count. Here the factorIndexList(1,:) specifies the sequence index while factorIndexList(2,:) specifies the tensor sequence index of the source element in the srcField. Further factorIndexList(3,:) specifies the sequence index and factorIndexList(4,:) specifies the tensor sequence index of the destination element in the dstField.

See section 28.2.18 for details on the definition of Field *sequence indices* and *tensor sequence indices*.

[ignoreUnmatchedIndices] A logical flag that affects the behavior for when sequence indices in the sparse matrix are encountered that do not have a match on the srcField or dstField side. The default setting is .false., indicating that it is an error when such a situation is encountered. Setting ignoreUnmatchedIndices to .true. ignores entries with unmatched indices.

[srcTermProcessing] The srcTermProcessing parameter controls how many source terms, located on the same PET and summing into the same destination element, are summed into partial sums on the source PET before being transferred to the destination PET. A value of 0 indicates that the entire arithmetic is done on the destination PET; source elements are neither multiplied by their factors nor added into partial sums before being sent off by the source PET. A value of 1 indicates that source elements are multiplied by their factors on the source side before being sent to the destination PET. Larger values of srcTermProcessing indicate the maximum number of terms in the partial sums on the source side.

Note that partial sums may lead to bit-for-bit differences in the results. See section **??** for an in-depth discussion of *all* bit-for-bit reproducibility aspects related to route-based communication methods.

The `ESMF_FieldSMMStore()` method implements an auto-tuning scheme for the `srcTermProcessing` parameter. The intent on the `srcTermProcessing` argument is "`inout`" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `srcTermProcessing` parameter, and the auto-tuning phase is skipped. In this case the `srcTermProcessing` argument is not modified on return. If the provided argument is < 0, the `srcTermProcessing` parameter is determined internally using the auto-tuning scheme. In this case the `srcTermProcessing` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `srcTermProcessing` argument is omitted.

[pipelineDepth] The `pipelineDepth` parameter controls how many messages a PET may have outstanding during a sparse matrix exchange. Larger values of `pipelineDepth` typically lead to better performance. However, on some systems too large a value may lead to performance degradation, or runtime errors.

Note that the pipeline depth has no effect on the bit-for-bit reproducibility of the results. However, it may affect the performance reproducibility of the exchange.

The `ESMF_FieldSMMStore()` method implements an auto-tuning scheme for the `pipelineDepth` parameter. The intent on the `pipelineDepth` argument is "`inout`" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `pipelineDepth` parameter, and the auto-tuning phase is skipped. In this case the `pipelineDepth` argument is not modified on return. If the provided argument is < 0, the `pipelineDepth` parameter is determined internally using the auto-tuning scheme. In this case the `pipelineDepth` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `pipelineDepth` argument is omitted.

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

26.6.70 ESMF_FieldSMMStore - Precompute Field sparse matrix multiplication without local factors

INTERFACE:

```
   ! Private name; call using ESMF_FieldSMMStore()
     subroutine ESMF_FieldSMMStoreNF(srcField, dstField, &
         routehandle, ignoreUnmatchedIndices, &
         srcTermProcessing, pipelineDepth, rc)
```

ARGUMENTS:

```
         type(ESMF_Field),       intent(in)              :: srcField
         type(ESMF_Field),       intent(inout)           :: dstField
         type(ESMF_RouteHandle), intent(inout)           :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
         logical,                intent(in),    optional :: ignoreUnmatchedIndices
         integer,                intent(inout), optional :: srcTermProcessing
         integer,                intent(inout), optional :: pipeLineDepth
         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:

  6.1.0 Added arguments `srcTermProcessing`, `pipelineDepth` The two arguments `srcTermProcessing` and `pipelineDepth` provide access to the tuning parameters affecting the sparse matrix execution.

  7.0.0 Added argument `transposeRoutehandle` to allow a handle to the transposed matrix operation to be returned.
  Added argument `ignoreUnmatchedIndices` to support sparse matrices that contain elements with indices that do not have a match within the source or destination Array.

  7.1.0r Removed argument `transposeRoutehandle` and provide it via interface overloading instead. This allows argument `srcField` to stay strictly intent(in) for this entry point.

DESCRIPTION:

Store a Field sparse matrix multiplication operation from `srcField` to `dstField`. PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* `factorList` and `factorIndexList` arguments.

Both `srcField` and `dstField` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 28.2.18 for details on the definition of *sequence indices*. SMM corresponds to an identity mapping of the source Field vector to the destination Field vector.

Source and destination Fields may be of different <type><kind>. Further source and destination Fields may differ in shape, however, the number of elements must match.

It is erroneous to specify the identical Field object for srcField and dstField arguments.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_FieldSMM()` on any pair of Fields that matches `srcField` and `dstField` in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This method is overloaded for:
`ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`,
`ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

This call is collective across the current VM.

For examples and associated documentation regarding this method see Section 26.3.33.

The arguments are:

srcField `ESMF_Field` with source data.

dstField `ESMF_Field` with destination data. The data in this Field may be destroyed by this call.

routehandle Handle to the precomputed Route.

[ignoreUnmatchedIndices] A logical flag that affects the behavior for when sequence indices in the sparse matrix are encountered that do not have a match on the `srcField` or `dstField` side. The default setting is `.false.`, indicating that it is an error when such a situation is encountered. Setting `ignoreUnmatchedIndices` to `.true.` ignores entries with unmatched indices.

[srcTermProcessing] The `srcTermProcessing` parameter controls how many source terms, located on the same PET and summing into the same destination element, are summed into partial sums on the source PET before being transferred to the destination PET. A value of 0 indicates that the entire arithmetic is done on the destination PET; source elements are neither multiplied by their factors nor added into partial sums before being sent off by the source PET. A value of 1 indicates that source elements are multiplied by their factors on the source side before being sent to the destination PET. Larger values of `srcTermProcessing` indicate the maximum number of terms in the partial sums on the source side.

Note that partial sums may lead to bit-for-bit differences in the results. See section **??** for an in-depth discussion of *all* bit-for-bit reproducibility aspects related to route-based communication methods.

The `ESMF_FieldSMMStore()` method implements an auto-tuning scheme for the `srcTermProcessing` parameter. The intent on the `srcTermProcessing` argument is "`inout`" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `srcTermProcessing` parameter, and the auto-tuning phase is skipped. In this case the `srcTermProcessing` argument is not modified on return. If the provided argument is < 0, the `srcTermProcessing` parameter is determined internally using the auto-tuning scheme. In this case the `srcTermProcessing` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `srcTermProcessing` argument is omitted.

[pipelineDepth] The `pipelineDepth` parameter controls how many messages a PET may have outstanding during a sparse matrix exchange. Larger values of `pipelineDepth` typically lead to better performance. However, on some systems too large a value may lead to performance degradation, or runtime errors.

Note that the pipeline depth has no effect on the bit-for-bit reproducibility of the results. However, it may affect the performance reproducibility of the exchange.

The `ESMF_FieldSMMStore()` method implements an auto-tuning scheme for the `pipelineDepth` parameter. The intent on the `pipelineDepth` argument is "`inout`" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `pipelineDepth` parameter, and the auto-tuning phase is skipped. In this case the `pipelineDepth` argument is not modified on return. If the provided argument is < 0, the `pipelineDepth` parameter is determined internally using the auto-tuning scheme. In this case the `pipelineDepth` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `pipelineDepth` argument is omitted.

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

26.6.71 ESMF_FieldSMMStore - Precompute Field sparse matrix multiplication and transpose without local factors

INTERFACE:

```
! Private name; call using ESMF_FieldSMMStore()
  subroutine ESMF_FieldSMMStoreNFTR(srcField, dstField, &
      routehandle, transposeRoutehandle, ignoreUnmatchedIndices, &
      srcTermProcessing, pipelineDepth, rc)
```

ARGUMENTS:

```
        type(ESMF_Field),       intent(inout)           :: srcField
        type(ESMF_Field),       intent(inout)           :: dstField
        type(ESMF_RouteHandle), intent(inout)           :: routehandle
        type(ESMF_RouteHandle), intent(inout)           :: transposeRoutehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
        logical,                intent(in),    optional :: ignoreUnmatchedIndices
        integer,                intent(inout), optional :: srcTermProcessing
        integer,                intent(inout), optional :: pipeLineDepth
        integer,                intent(out),   optional :: rc
```

DESCRIPTION:

Store a Field sparse matrix multiplication operation from `srcField` to `dstField`. PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* `factorList` and `factorIndexList` arguments.

Both `srcField` and `dstField` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 28.2.18 for details on the definition of *sequence indices*. SMM corresponds to an identity mapping of the source Field vector to the destination Field vector.

Source and destination Fields may be of different <type><kind>. Further source and destination Fields may differ in shape, however, the number of elements must match.

It is erroneous to specify the identical Field object for srcField and dstField arguments.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_FieldSMM()` on any pair of Fields that matches `srcField` and `dstField` in *type*, *kind*, and memory layout of the *gridded* dimensions. However, the size, number, and index order of *ungridded* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This method is overloaded for:
`ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`,
`ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

This call is collective across the current VM.

For examples and associated documentation regarding this method see Section 26.3.33.

The arguments are:

srcField `ESMF_Field` with source data. The data in this Field may be destroyed by this call.

dstField `ESMF_Field` with destination data. The data in this Field may be destroyed by this call.

routehandle Handle to the precomputed Route.

transposeRoutehandle A handle to the transposed matrix operation is returned. The transposed operation goes from `dstArray` to `srcArray`.

[ignoreUnmatchedIndices] A logical flag that affects the behavior for when sequence indices in the sparse matrix are encountered that do not have a match on the `srcField` or `dstField` side. The default setting is `.false.`, indicating that it is an error when such a situation is encountered. Setting `ignoreUnmatchedIndices` to `.true.` ignores entries with unmatched indices.

[srcTermProcessing] The `srcTermProcessing` parameter controls how many source terms, located on the same PET and summing into the same destination element, are summed into partial sums on the source PET before being transferred to the destination PET. A value of 0 indicates that the entire arithmetic is done on the destination PET; source elements are neither multiplied by their factors nor added into partial sums before being sent off by the source PET. A value of 1 indicates that source elements are multiplied by their factors on the source side before being sent to the destination PET. Larger values of `srcTermProcessing` indicate the maximum number of terms in the partial sums on the source side.

Note that partial sums may lead to bit-for-bit differences in the results. See section **??** for an in-depth discussion of *all* bit-for-bit reproducibility aspects related to route-based communication methods.

The `ESMF_FieldSMMStore()` method implements an auto-tuning scheme for the `srcTermProcessing` parameter. The intent on the `srcTermProcessing` argument is "`inout`" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `srcTermProcessing` parameter, and the auto-tuning phase is skipped. In this case the `srcTermProcessing` argument is not modified on return. If the provided argument is < 0, the `srcTermProcessing` parameter is determined internally using the auto-tuning scheme. In this case the `srcTermProcessing` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `srcTermProcessing` argument is omitted.

[pipelineDepth] The `pipelineDepth` parameter controls how many messages a PET may have outstanding during a sparse matrix exchange. Larger values of `pipelineDepth` typically lead to better performance. However, on some systems too large a value may lead to performance degradation, or runtime errors.

Note that the pipeline depth has no effect on the bit-for-bit reproducibility of the results. However, it may affect the performance reproducibility of the exchange.

The `ESMF_FieldSMMStore()` method implements an auto-tuning scheme for the `pipelineDepth` parameter. The intent on the `pipelineDepth` argument is "`inout`" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `pipelineDepth` parameter, and the auto-tuning phase is skipped. In this case the `pipelineDepth` argument is not modified on return. If the provided argument is < 0, the `pipelineDepth` parameter is determined internally using the auto-tuning scheme. In this case the `pipelineDepth` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `pipelineDepth` argument is omitted.

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

26.6.72 ESMF_FieldSMMStore - Precompute sparse matrix multiplication using factors read from file

INTERFACE:

```
   ! Private name; call using ESMF_FieldSMMStore()
     subroutine ESMF_FieldSMMStoreFromFile(srcField, dstField, filename, &
       routehandle, ignoreUnmatchedIndices, &
       srcTermProcessing, pipelineDepth, rc)

   ! ARGUMENTS:
       type(ESMF_Field),       intent(in)              :: srcField
       type(ESMF_Field),       intent(inout)           :: dstField
       character(len=*),       intent(in)              :: filename
       type(ESMF_RouteHandle), intent(inout)           :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       logical,                intent(in),    optional :: ignoreUnmatchedIndices
       integer,                intent(inout), optional :: srcTermProcessing
```

```
    integer,                intent(inout), optional :: pipeLineDepth
    integer,                intent(out),   optional :: rc
```

DESCRIPTION:

Compute an ESMF_RouteHandle using factors read from file.

The arguments are:

srcField ESMF_Field with source data.

dstField ESMF_Field with destination data. The data in this Field may be destroyed by this call.

filename Path to the file containing weights for creating an ESMF_RouteHandle. See (12.9) for a description of the SCRIP weight file format. Only "row", "col", and "S" variables are required. They must be one-dimensionsal with dimension "n_s".

routehandle Handle to the ESMF_RouteHandle.

[ignoreUnmatchedIndices] A logical flag that affects the behavior for when sequence indices in the sparse matrix are encountered that do not have a match on the srcField or dstField side. The default setting is .false., indicating that it is an error when such a situation is encountered. Setting ignoreUnmatchedIndices to .true. ignores entries with unmatched indices.

[srcTermProcessing] The srcTermProcessing parameter controls how many source terms, located on the same PET and summing into the same destination element, are summed into partial sums on the source PET before being transferred to the destination PET. A value of 0 indicates that the entire arithmetic is done on the destination PET; source elements are neither multiplied by their factors nor added into partial sums before being sent off by the source PET. A value of 1 indicates that source elements are multiplied by their factors on the source side before being sent to the destination PET. Larger values of srcTermProcessing indicate the maximum number of terms in the partial sums on the source side.

Note that partial sums may lead to bit-for-bit differences in the results. See section **??** for an in-depth discussion of *all* bit-for-bit reproducibility aspects related to route-based communication methods.

The ESMF_FieldSMMStore() method implements an auto-tuning scheme for the srcTermProcessing parameter. The intent on the srcTermProcessing argument is "inout" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the srcTermProcessing parameter, and the auto-tuning phase is skipped. In this case the srcTermProcessing argument is not modified on return. If the provided argument is < 0, the srcTermProcessing parameter is determined internally using the auto-tuning scheme. In this case the srcTermProcessing argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional srcTermProcessing argument is omitted.

[pipelineDepth] The pipelineDepth parameter controls how many messages a PET may have outstanding during a sparse matrix exchange. Larger values of pipelineDepth typically lead to better performance. However, on some systems too large a value may lead to performance degradation, or runtime errors.

Note that the pipeline depth has no effect on the bit-for-bit reproducibility of the results. However, it may affect the performance reproducibility of the exchange. The ESMF_FieldSMMStore() method implements an auto-tuning scheme for the pipelineDepth parameter. The intent on the pipelineDepth argument is "inout" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the pipelineDepth parameter, and the auto-tuning phase is skipped. In this case the pipelineDepth argument is not modified on return. If the provided argument is < 0, the pipelineDepth parameter is determined internally using the auto-tuning scheme. In this case the pipelineDepth argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional pipelineDepth argument is omitted.

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

26.6.73 ESMF_FieldSMMStore - Precompute sparse matrix multiplication and transpose using factors read from file

INTERFACE:

```
   ! Private name; call using ESMF_FieldSMMStore()
     subroutine ESMF_FieldSMMStoreFromFileTR(srcField, dstField, filename, &
       routehandle, transposeRoutehandle, &
       ignoreUnmatchedIndices, srcTermProcessing, pipelineDepth, rc)

   ! ARGUMENTS:
       type(ESMF_Field),       intent(inout)           :: srcField
       type(ESMF_Field),       intent(inout)           :: dstField
       character(len=*),       intent(in)              :: filename
       type(ESMF_RouteHandle), intent(inout)           :: routehandle
       type(ESMF_RouteHandle), intent(inout)           :: transposeRoutehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
       logical,                intent(in),    optional :: ignoreUnmatchedIndices
       integer,                intent(inout), optional :: srcTermProcessing
       integer,                intent(inout), optional :: pipeLineDepth
       integer,                intent(out),   optional :: rc
```

DESCRIPTION:

Compute an `ESMF_RouteHandle` using factors read from file.

The arguments are:

srcField `ESMF_Field` with source data. The data in this Array may be destroyed by this call.

dstField `ESMF_Field` with destination data. The data in this Field may be destroyed by this call.

filename Path to the file containing weights for creating an `ESMF_RouteHandle`. See (12.9) for a description of the SCRIP weight file format. Only "row", "col", and "S" variables are required. They must be one-dimensionsal with dimension "n_s".

routehandle Handle to the `ESMF_RouteHandle`.

transposeRoutehandle A handle to the transposed matrix operation is returned. The transposed operation goes from `dstArray` to `srcArray`.

[ignoreUnmatchedIndices] A logical flag that affects the behavior for when sequence indices in the sparse matrix are encountered that do not have a match on the `srcField` or `dstField` side. The default setting is `.false.`, indicating that it is an error when such a situation is encountered. Setting `ignoreUnmatchedIndices` to `.true.` ignores entries with unmatched indices.

[srcTermProcessing] The `srcTermProcessing` parameter controls how many source terms, located on the same PET and summing into the same destination element, are summed into partial sums on the source PET before being transferred to the destination PET. A value of 0 indicates that the entire arithmetic is done on the destination PET; source elements are neither multiplied by their factors nor added into partial sums before being sent off by the source PET. A value of 1 indicates that source elements are multiplied by their factors on the source

side before being sent to the destination PET. Larger values of srcTermProcessing indicate the maximum number of terms in the partial sums on the source side.

Note that partial sums may lead to bit-for-bit differences in the results. See section **??** for an in-depth discussion of *all* bit-for-bit reproducibility aspects related to route-based communication methods.

The ESMF_FieldSMMStore() method implements an auto-tuning scheme for the srcTermProcessing parameter. The intent on the srcTermProcessing argument is "inout" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the srcTermProcessing parameter, and the auto-tuning phase is skipped. In this case the srcTermProcessing argument is not modified on return. If the provided argument is < 0, the srcTermProcessing parameter is determined internally using the auto-tuning scheme. In this case the srcTermProcessing argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional srcTermProcessing argument is omitted.

[pipelineDepth] The pipelineDepth parameter controls how many messages a PET may have outstanding during a sparse matrix exchange. Larger values of pipelineDepth typically lead to better performance. However, on some systems too large a value may lead to performance degradation, or runtime errors.

Note that the pipeline depth has no effect on the bit-for-bit reproducibility of the results. However, it may affect the performance reproducibility of the exchange. The ESMF_FieldSMMStore() method implements an auto-tuning scheme for the pipelineDepth parameter. The intent on the pipelineDepth argument is "inout" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the pipelineDepth parameter, and the auto-tuning phase is skipped. In this case the pipelineDepth argument is not modified on return. If the provided argument is < 0, the pipelineDepth parameter is determined internally using the auto-tuning scheme. In this case the pipelineDepth argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional pipelineDepth argument is omitted.

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

26.6.74 ESMF_FieldValidate - Check validity of a Field

INTERFACE:

```
subroutine ESMF_FieldValidate(field, rc)
```

ARGUMENTS:

```
type(ESMF_Field), intent(in)            :: field
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
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:

Validates that the field is internally consistent. Currently this method determines if the field is uninitialized or already destroyed. It validates the contained array and grid objects. The code also checks if the array and grid sizes

agree. This check compares the distgrid contained in array and grid; then it proceeds to compare the computational bounds contained in array and grid.

The method returns an error code if problems are found.

The arguments are:

field `ESMF_Field` to validate.

[rc] Return code; equals `ESMF_SUCCESS` if the `field` is valid.

26.6.75 ESMF_FieldWrite - Write Field data into a file

INTERFACE:

```
   subroutine ESMF_FieldWrite(field, fileName,   &
       variableName, convention, purpose, overwrite, status, timeslice, iofmt, rc)
```

ARGUMENTS:

```
     type(ESMF_Field),           intent(in)             :: field
     character(*),               intent(in)             :: fileName
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     character(*),               intent(in),  optional  :: variableName
     character(*),               intent(in),  optional  :: convention
     character(*),               intent(in),  optional  :: purpose
     logical,                    intent(in),  optional  :: overwrite
     type(ESMF_FileStatus_Flag), intent(in),  optional  :: status
     integer,                    intent(in),  optional  :: timeslice
     type(ESMF_IOFmt_Flag),      intent(in),  optional  :: iofmt
     integer,                    intent(out), optional  :: rc
```

DESCRIPTION:

Write Field data into a file. For this API to be functional, the environment variable `ESMF_PIO` should be set to "internal" when the ESMF library is built. Please see the section on Data I/O, **??**.

When `convention` and `purpose` arguments are specified, a NetCDF variable can be created with user-specified dimension labels and attributes. Dimension labels may be defined for both gridded and ungridded dimensions. Dimension labels for gridded dimensions are specified at the Grid level by attaching an ESMF Attribute package to it. The Attribute package must contain an attribute named by the pre-defined ESMF parameter `ESMF_ATT_GRIDDED_DIM_LABELS`. The corresponding value is an array of character strings specifying the desired names of the dimensions. Likewise, for ungridded dimensions, an Attribute package is attached at the Field level. The name of the name must be `ESMF_ATT_UNGRIDDED_DIM_LABELS`.

NetCDF attributes for the variable can also be specified. As with dimension labels, an Attribute package is added to the Field with the desired names and values. A value may be either a scalar character string, or a scalar or array of type integer, real, or double precision. Dimension label attributes can co-exist with variable attributes within a common Attribute package.

Limitations:

- Only single tile Fields are supported.

- Not supported in ESMF_COMM=mpiuni mode.

The arguments are:

field The ESMF_Field object that contains data to be written.

fileName The name of the output file to which Field data is written.

[variableName] Variable name in the output file; default is the "name" of field. Use this argument only in the I/O format (such as NetCDF) that supports variable name. If the I/O format does not support this (such as binary format), ESMF will return an error code.

[convention] Specifies an Attribute package associated with the Field, used to create NetCDF dimension labels and attributes for the variable in the file. When this argument is present, the purpose argument must also be present. Use this argument only with a NetCDF I/O format. If binary format is used, ESMF will return an error code.

[purpose] Specifies an Attribute package associated with the Field, used to create NetCDF dimension labels and attributes for the variable in the file. When this argument is present, the convention argument must also be present. Use this argument only with a NetCDF I/O format. If binary format is used, ESMF will return an error code.

[overwrite] A logical flag, the default is .false., i.e., existing field data may *not* be overwritten. If .true., the overwrite behavior depends on the value of iofmt as shown below:

iofmt **=** ESMF_IOFMT_BIN**:** All data in the file will be overwritten with each field's data.

iofmt **=** ESMF_IOFMT_NETCDF, ESMF_IOFMT_NETCDF_64BIT_OFFSET**:** Only the data corresponding to each field's name will be be overwritten. If the timeslice option is given, only data for the given timeslice may be overwritten. Note that it is always an error to attempt to overwrite a NetCDF variable with data which has a different shape.

[status] The file status. Please see Section **??** for the list of options. If not present, defaults to ESMF_FILESTATUS_UNKNOWN.

[timeslice] Some I/O formats (e.g. NetCDF) support the output of data in form of time slices. An unlimited dimension called time is defined in the file variable for this capability. The timeslice argument provides access to the time dimension, and must have a positive value. The behavior of this option may depend on the setting of the overwrite flag:

overwrite = .false.**:** If the timeslice value is less than the maximum time already in the file, the write will fail.

overwrite = .true.**:** Any positive timeslice value is valid.

By default, i.e. by omitting the timeslice argument, no provisions for time slicing are made in the output file, however, if the file already contains a time axis for the variable, a timeslice one greater than the maximum will be written.

[iofmt] The I/O format. Please see Section **??** for the list of options. If not present, file names with a .bin extension will use ESMF_IOFMT_BIN, and file names with a .nc extension will use ESMF_IOFMT_NETCDF. Other files default to ESMF_IOFMT_NETCDF.

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

26.7 Class API: Field Utilities

26.7.1 ESMF_GridGetFieldBounds - Get precomputed DE-local Fortran data array bounds for creating a Field from a Grid and Fortran array

INTERFACE:

```
subroutine ESMF_GridGetFieldBounds(grid, &
    localDe, staggerloc, gridToFieldMap, &
    ungriddedLBound, ungriddedUBound, &
    totalLWidth, totalUWidth, &
    totalLBound, totalUBound, totalCount, rc)
```

ARGUMENTS:

```
    type(ESMF_Grid),       intent(in)            :: grid
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer,               intent(in),  optional :: localDe
    type(ESMF_StaggerLoc), intent(in),  optional :: staggerloc
    integer,               intent(in),  optional :: gridToFieldMap(:)
    integer,               intent(in),  optional :: ungriddedLBound(:)
    integer,               intent(in),  optional :: ungriddedUBound(:)
    integer,               intent(in),  optional :: totalLWidth(:)
    integer,               intent(in),  optional :: totalUWidth(:)
    integer,               intent(out), optional :: totalLBound(:)
    integer,               intent(out), optional :: totalUBound(:)
    integer,               intent(out), optional :: totalCount(:)
    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:

Compute the lower and upper bounds of Fortran data array that can later be used in FieldCreate interface to create a `ESMF_Field` from a `ESMF_Grid` and the Fortran data array. For an example and associated documentation using this method see section 26.3.9.

The arguments are:

grid `ESMF_Grid`.

[localDe] Local DE for which information is requested. `[0,..,localDeCount-1]`. For `localDeCount==1` the `localDe` argument may be omitted, in which case it will default to `localDe=0`.

[staggerloc] Stagger location of data in grid cells. For valid predefined values and interpretation of results see section 31.2.6.

[gridToFieldMap] List with number of elements equal to the `grid`ls dimCount. The list elements map each dimension of the `grid` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map all of the `grid`ls dimensions against the lowest dimensions of the `field` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `field` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the `field`.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[totalLWidth] Lower bound of halo region. The size of this array is the number of dimensions in the `grid`. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalLWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should be max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[totalUWidth] Upper bound of halo region. The size of this array is the number of dimensions in the `grid`. However, ordering of the elements needs to be the same as they appear in the `field`. Values default to 0. If values for totalUWidth are specified they must be reflected in the size of the `field`. That is, for each gridded dimension the `field` size should max(`totalLWidth` + `totalUWidth` + `computationalCount`, `exclusiveCount`).

[totalLBound] The relative lower bounds of Fortran data array to be used later in `ESMF_FieldCreate` from `ESMF_Grid` and Fortran data array. This is an output variable from this user interface.

The relative lower bounds of Fortran data array to be used

[totalUBound] The relative upper bounds of Fortran data array to be used later in `ESMF_FieldCreate` from `ESMF_Grid` and Fortran data array. This is an output variable from this user interface.

[totalCount] Number of elements need to be allocated for Fortran data array to be used later in `ESMF_FieldCreate` from `ESMF_Grid` and Fortran data array. This is an output variable from this user interface.

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

26.7.2 ESMF_LocStreamGetFieldBounds - Get precomputed DE-local Fortran data array bounds for creating a Field from a LocStream and Fortran array

INTERFACE:

```
subroutine ESMF_LocStreamGetFieldBounds(locstream, &
    localDe, gridToFieldMap, &
    ungriddedLBound, ungriddedUBound, &
```

```
    totalLBound, totalUBound, totalCount, rc)
```

ARGUMENTS:

```
     type(ESMF_LocStream), intent(in)            :: locstream
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,              intent(in),  optional :: localDe
     integer,              intent(in),  optional :: gridToFieldMap(:)
     integer,              intent(in),  optional :: ungriddedLBound(:)
     integer,              intent(in),  optional :: ungriddedUBound(:)
     integer,              intent(out), optional :: totalLBound(:)
     integer,              intent(out), optional :: totalUBound(:)
     integer,              intent(out), optional :: totalCount(:)
     integer,              intent(out), optional :: rc
```

DESCRIPTION:

Compute the lower and upper bounds of Fortran data array that can later be used in FieldCreate interface to create a `ESMF_Field` from a `ESMF_LocStream` and the Fortran data array. For an example and associated documentation using this method see section 26.3.9.

The arguments are:

locstream `ESMF_LocStream`.

[localDe] Local DE for which information is requested. `[0,..,localDeCount-1]`. For `localDeCount==1` the `localDe` argument may be omitted, in which case it will default to `localDe=0`.

[gridToFieldMap] List with number of elements equal to 1. The list elements map the dimension of the `locstream` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map the `locstream`s dimension against the lowest dimension of the `field` in sequence, i.e. `gridToFieldMap` = (/1/). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `field` rank. The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the field.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than 1, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than 1, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[totalLBound] The relative lower bounds of Fortran data array to be used later in `ESMF_FieldCreate` from `ESMF_LocStream` and Fortran data array. This is an output variable from this user interface.

[totalUBound] The relative upper bounds of Fortran data array to be used later in `ESMF_FieldCreate` from `ESMF_LocStream` and Fortran data array. This is an output variable from this user interface.

[totalCount] Number of elements need to be allocated for Fortran data array to be used later in ESMF_FieldCreate from ESMF_LocStream and Fortran data array. This is an output variable from this user interface.

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

26.7.3 ESMF_MeshGetFieldBounds - Get precomputed DE-local Fortran data array bounds for creating a Field from a Mesh and a Fortran array

INTERFACE:

```
subroutine ESMF_MeshGetFieldBounds(mesh, &
    meshloc, &
    localDe, gridToFieldMap, &
    ungriddedLBound, ungriddedUBound, &
    totalLBound, totalUBound, totalCount, rc)
```

ARGUMENTS:

```
type(ESMF_Mesh), intent(in)            :: mesh
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_MeshLoc),intent(in),optional :: meshloc
integer,         intent(in),  optional :: localDe
integer,         intent(in),  optional :: gridToFieldMap(:)
integer,         intent(in),  optional :: ungriddedLBound(:)
integer,         intent(in),  optional :: ungriddedUBound(:)
integer,         intent(out), optional :: totalLBound(:)
integer,         intent(out), optional :: totalUBound(:)
integer,         intent(out), optional :: totalCount(:)
integer,         intent(out), optional :: rc
```

DESCRIPTION:

Compute the lower and upper bounds of Fortran data array that can later be used in FieldCreate interface to create a ESMF_Field from a ESMF_Mesh and the Fortran data array. For an example and associated documentation using this method see section 26.3.9.

The arguments are:

mesh ESMF_Mesh.

[meshloc] Which part of the mesh to build the Field on. Can be set to either ESMF_MESHLOC_NODE or ESMF_MESHLOC_ELEMENT. If not set, defaults to ESMF_MESHLOC_NODE.

[localDe] Local DE for which information is requested. [0,..,localDeCount-1]. For localDeCount==1 the localDe argument may be omitted, in which case it will default to localDe=0.

[gridToFieldMap] List with number of elements equal to the `grid`ls dimCount. The list elements map each dimension of the `grid` to a dimension in the `field` by specifying the appropriate `field` dimension index. The default is to map all of the `grid`ls dimensions against the lowest dimensions of the `field` in sequence, i.e. `gridToFieldMap` = (/1,2,3,.../). The values of all `gridToFieldMap` entries must be greater than or equal to one and smaller than or equal to the `field` rank. It is erroneous to specify the same `gridToFieldMap` entry multiple times. The total ungridded dimensions in the `field` are the total `field` dimensions less the dimensions in the `grid`. Ungridded dimensions must be in the same order they are stored in the field.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedLBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the `field`. The number of elements in the `ungriddedUBound` is equal to the number of ungridded dimensions in the `field`. All ungridded dimensions of the `field` are also undistributed. When field dimension count is greater than grid dimension count, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the `field`.

[totalLBound] The relative lower bounds of Fortran data array to be used later in `ESMF_FieldCreate` from `ESMF_Mesh` and Fortran data array. This is an output variable from this user interface.

[totalUBound] The relative upper bounds of Fortran data array to be used later in `ESMF_FieldCreate` from `ESMF_Mesh` and Fortran data array. This is an output variable from this user interface.

[totalCount] Number of elements need to be allocated for Fortran data array to be used later in `ESMF_FieldCreate` from `ESMF_Mesh` and Fortran data array. This is an output variable from this user interface.

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

26.7.4 ESMF_XGridGetFieldBounds - Get precomputed DE-local Fortran data array bounds for creating a Field from an XGrid and a Fortran array

INTERFACE:

```
subroutine ESMF_XGridGetFieldBounds(xgrid, &
    xgridside, gridindex, localDe, gridToFieldMap, &
    ungriddedLBound, ungriddedUBound, &
    totalLBound, totalUBound, totalCount, rc)
```

ARGUMENTS:

```
type(ESMF_XGrid),          intent(in)            :: xgrid
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_XGridSide_Flag), intent(in),  optional :: xgridside
integer,                   intent(in),  optional :: gridindex
integer,                   intent(in),  optional :: localDe
integer,                   intent(in),  optional :: gridToFieldMap(:)
integer,                   intent(in),  optional :: ungriddedLBound(:)
```

```
integer,                    intent(in),  optional :: ungriddedUBound(:)
integer,                    intent(out), optional :: totalLBound(:)
integer,                    intent(out), optional :: totalUBound(:)
integer,                    intent(out), optional :: totalCount(:)
integer,                    intent(out), optional :: rc
```

DESCRIPTION:

Compute the lower and upper bounds of Fortran data array that can later be used in FieldCreate interface to create a ESMF_Field from a ESMF_XGrid and the Fortran data array. For an example and associated documentation using this method see section 26.3.9.

The arguments are:

xgrid ESMF_XGrid object.

[xgridside] Which side of the XGrid to create the Field on (either ESMF_XGRIDSIDE_A, ESMF_XGRIDSIDE_B, or ESMF_XGRIDSIDE_BALANCED). If not passed in then defaults to ESMF_XGRIDSIDE_BALANCED.

[gridindex] If xgridside is ESMF_XGRIDSIDE_A or ESMF_XGRIDSIDE_B then this index tells which Grid on that side to create the Field on. If not provided, defaults to 1.

[localDe] Local DE for which information is requested. [0,..,localDeCount-1]. For localDeCount==1 the localDe argument may be omitted, in which case it will default to localDe=0.

[gridToFieldMap] List with number of elements equal to 1. The list elements map the dimension of the locstream to a dimension in the field by specifying the appropriate field dimension index. The default is to map the locstreamls dimension against the lowest dimension of the field in sequence, i.e. gridToFieldMap = (/1/). The values of all gridToFieldMap entries must be greater than or equal to one and smaller than or equal to the field rank. The total ungridded dimensions in the field are the total field dimensions less the dimensions in the grid. Ungridded dimensions must be in the same order they are stored in the field.

[ungriddedLBound] Lower bounds of the ungridded dimensions of the field. The number of elements in the ungriddedLBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than 1, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[ungriddedUBound] Upper bounds of the ungridded dimensions of the field. The number of elements in the ungriddedUBound is equal to the number of ungridded dimensions in the field. All ungridded dimensions of the field are also undistributed. When field dimension count is greater than 1, both ungriddedLBound and ungriddedUBound must be specified. When both are specified the values are checked for consistency. Note that the the ordering of these ungridded dimensions is the same as their order in the field.

[totalLBound] The relative lower bounds of Fortran data array to be used later in ESMF_FieldCreate from ESMF_LocStream and Fortran data array. This is an output variable from this user interface.

[totalUBound] The relative upper bounds of Fortran data array to be used later in ESMF_FieldCreate from ESMF_LocStream and Fortran data array. This is an output variable from this user interface.

[totalCount] Number of elements need to be allocated for Fortran data array to be used later in ESMF_FieldCreate from ESMF_LocStream and Fortran data array. This is an output variable from this user interface.

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

27 ArrayBundle Class

27.1 Description

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

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

27.2 Use and Examples

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

27.2.1 Creating an ArrayBundle from a list of Arrays

An ArrayBundle is created from a list of ESMF_Array objects.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)


distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
  regDecomp=(/2,3/), rc=rc)


allocate(arrayList(2))
arrayList(1) = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
               rc=rc)


arrayList(2) = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
               rc=rc)
```

Now arrayList is used to create an ArrayBundle object.

```
arraybundle = ESMF_ArrayBundleCreate(arrayList=arrayList, &
  name="MyArrayBundle", rc=rc)
```

Here the temporary arrayList can be deallocated. This will not affect the ESMF Array objects inside the ArrayBundle. However, the Array objects must not be deallocated while the ArrayBundle references them.

```
deallocate(arrayList)
```

27.2.2 Adding, removing, replacing Arrays in the ArrayBundle

Individual Arrays can be added using the Fortran array constructor syntax `(/ ... /)`. Here an ESMF_Array is created on the fly and immediately added to the ArrayBundle.

```
call ESMF_ArrayBundleAdd(arraybundle, arrayList=(/ &
  ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, name="AonFly")/), &
  rc=rc)
```

Items in the ArrayBundle can be replaced by items with the same name.

```
call ESMF_ArraySpecSet(arrayspec2, typekind=ESMF_TYPEKIND_R4, rank=2, rc=rc)


call ESMF_ArrayBundleReplace(arraybundle, arrayList=(/ &
  ESMF_ArrayCreate(arrayspec=arrayspec2, distgrid=distgrid, name="AonFly")/), &
  rc=rc)
```

Items can be removed from the ArrayBundle by providing their name.

```
call ESMF_ArrayBundleRemove(arraybundle, arrayNameList=(/"AonFly"/), rc=rc)
```

The ArrayBundle AddReplace() method can be used to conveniently add an item to the ArrayBundle, or replacing an existing item of the same name.

```
call ESMF_ArrayBundleAddReplace(arraybundle, arrayList=(/ &
  ESMF_ArrayCreate(arrayspec=arrayspec2, distgrid=distgrid, name="AonFly")/), &
  rc=rc)
```

The ArrayBundle object can be printed at any time to list its contents by name.

```
call ESMF_ArrayBundlePrint(arraybundle, rc=rc)
```

27.2.3 Accessing Arrays inside the ArrayBundle

Individual items in the ArrayBundle can be accessed directly by their name.

```
call ESMF_ArrayBundleGet(arraybundle, arrayName="AonFly", array=arrayOut, &
  rc=rc)
```

A list containing all of the Arrays in the ArrayBundle can also be requested in a single call. This requires that a large enough list argument is passed into the `ESMF_ArrayBundleGet()` method. The exact number of items in the ArrayBundle can be queried using the `arrayCount` argument first.

```
call ESMF_ArrayBundleGet(arraybundle, arrayCount=arrayCount, rc=rc)
```

Then use `arrayCount` to correctly allocate the `arrayList` variable for a second call to `ESMF_ArrayBundleGet()`.

```
allocate(arrayList(arrayCount))
call ESMF_ArrayBundleGet(arraybundle, arrayList=arrayList, rc=rc)
```

Now the `arrayList` variable can be used to access the individual Arrays, e.g. to print them.

```
do i=1, arrayCount
  call ESMF_ArrayPrint(arrayList(i), rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
enddo
```

By default the `arrayList` returned by `ESMF_ArrayBundleGet()` contains the items in alphabetical order. To instead return the items in the same order in which they were added to the ArrayBundle, the `itemorderflag` argument is passed with a value of `ESMF_ITEMORDER_ADDORDER`.

```
call ESMF_ArrayBundleGet(arraybundle, arrayList=arrayList, &
  itemorderflag=ESMF_ITEMORDER_ADDORDER, rc=rc)
```

27.2.4 Destroying an ArrayBundle and its constituents

Destroying an ArrayBundle does not destroy the Arrays. In fact, it leaves the Arrays totally unchanged.

```
call ESMF_ArrayBundleDestroy(arraybundle, rc=rc)
```

The Arrays must be destroyed separately.

```
call ESMF_ArrayDestroy(arrayList(1), rc=rc)


call ESMF_ArrayDestroy(arrayList(2), rc=rc)


deallocate(arrayList)

call ESMF_DistGridDestroy(distgrid, rc=rc)
```

27.2.5 Halo communication

One of the most fundamental communication pattern in domain decomposition codes is the *halo* operation. The ESMF Array class supports halos by allowing memory for extra elements to be allocated on each DE. See section 28.2.15 for a discussion of the Array level halo operation. The ArrayBundle level extents the Array halo operation to bundles of Arrays.

First create an `ESMF_ArrayBundle` object containing a set of ESMF Arrays.

```
arraybundle = ESMF_ArrayBundleCreate(arrayList=arrayList, &
  name="MyArrayBundle", rc=rc)
```

The ArrayBundle object can be treated as a single entity. The `ESMF_ArrayBundleHaloStore()` call determines the most efficient halo exchange pattern for *all* Arrays that are part of `arraybundle`.

```
call ESMF_ArrayBundleHaloStore(arraybundle=arraybundle, &
  routehandle=haloHandle, rc=rc)
```

The halo exchange pattern stored in `haloHandle` can now be applied to the `arraybundle` object, or any other ArrayBundle that is compatible to the one used during the `ESMF_ArrayBundleHaloStore()` call.

```
call ESMF_ArrayBundleHalo(arraybundle=arraybundle, routehandle=haloHandle, &
  rc=rc)
```

Finally, when no longer needed, the resources held by `haloHandle` need to be returned to the system by calling `ESMF_ArrayBundleHaloRelease()`.

```
call ESMF_ArrayBundleHaloRelease(routehandle=haloHandle, rc=rc)
```

Finally the ArrayBundle object can be destroyed.

```
call ESMF_ArrayBundleDestroy(arraybundle, rc=rc)
```

27.3 Restrictions and Future Work

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

27.4 Design and Implementation Notes

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

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

27.5 Class API

27.5.1 ESMF_ArrayBundleAssignment(=) - ArrayBundle assignment

INTERFACE:

```
interface assignment(=)
arraybundle1 = arraybundle2
```

ARGUMENTS:

```
type(ESMF_ArrayBundle) :: arraybundle1
type(ESMF_ArrayBundle) :: arraybundle2
```

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:

Assign arraybundle1 as an alias to the same ESMF ArrayBundle object in memory as arraybundle2. If arraybundle2 is invalid, then arraybundle1 will be equally invalid after the assignment.

The arguments are:

arraybundle1 The `ESMF_ArrayBundle` object on the left hand side of the assignment.

arraybundle2 The `ESMF_ArrayBundle` object on the right hand side of the assignment.

27.5.2 ESMF_ArrayBundleOperator(==) - ArrayBundle equality operator

INTERFACE:

```
interface operator(==)
  if (arraybundle1 == arraybundle2) then ... endif
            OR
  result = (arraybundle1 == arraybundle2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_ArrayBundle), intent(in) :: arraybundle1
type(ESMF_ArrayBundle), intent(in) :: arraybundle2
```

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:

Test whether arraybundle1 and arraybundle2 are valid aliases to the same ESMF ArrayBundle object in memory. For a more general comparison of two ESMF ArrayBundles, going beyond the simple alias test, the ESMF_ArrayBundleMatch() function (not yet implemented) must be used.

The arguments are:

arraybundle1 The `ESMF_ArrayBundle` object on the left hand side of the equality operation.

arraybundle2 The `ESMF_ArrayBundle` object on the right hand side of the equality operation.

27.5.3 ESMF_ArrayBundleOperator(/=) - ArrayBundle not equal operator

INTERFACE:

```
interface operator(/=)
  if (arraybundle1 /= arraybundle2) then ... endif
            OR
  result = (arraybundle1 /= arraybundle2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_ArrayBundle), intent(in) :: arraybundle1
type(ESMF_ArrayBundle), intent(in) :: arraybundle2
```

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:

Test whether arraybundle1 and arraybundle2 are *not* valid aliases to the same ESMF ArrayBundle object in memory. For a more general comparison of two ESMF ArrayBundles, going beyond the simple alias test, the ESMF_ArrayBundleMatch() function (not yet implemented) must be used.

The arguments are:

arraybundle1 The ESMF_ArrayBundle object on the left hand side of the non-equality operation.

arraybundle2 The ESMF_ArrayBundle object on the right hand side of the non-equality operation.

27.5.4 ESMF_ArrayBundleAdd - Add Arrays to an ArrayBundle

INTERFACE:

```
     subroutine ESMF_ArrayBundleAdd(arraybundle, arrayList, &
       multiflag, relaxedflag, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle), intent(inout)         :: arraybundle
     type(ESMF_Array),       intent(in)            :: arrayList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),  optional :: multiflag
     logical,                intent(in),  optional :: relaxedflag
     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:

Add Array(s) to an ArrayBundle. It is an error if arrayList contains Arrays that match by name Arrays already contained in arraybundle.

arraybundle ESMF_ArrayBundle to be added to.

arrayList List of ESMF_Array objects to be added.

[multiflag] A setting of .true. allows multiple items with the same name to be added to arraybundle. For .false. added items must have unique names. The default setting is .false..

[relaxedflag] A setting of .true. indicates a relaxed definition of "add" under multiflag=.false. mode, where it is *not* an error if arrayList contains items with names that are also found in arraybundle. The arraybundle is left unchanged for these items. For .false. this is treated as an error condition. The default setting is .false..

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

27.5.5 ESMF_ArrayBundleAddReplace - Conditionally add or replace Arrays in an ArrayBundle

INTERFACE:

```
    subroutine ESMF_ArrayBundleAddReplace(arraybundle, arrayList, rc)
```

ARGUMENTS:

```
    type(ESMF_ArrayBundle), intent(inout)         :: arraybundle
    type(ESMF_Array),       intent(in)            :: arrayList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    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:

Arrays in `arrayList` that do not match any Arrays by name in `arraybundle` are added to the ArrayBundle. Arrays in `arraybundle` that match by name Arrays in `arrayList` are replaced by those Arrays.

arraybundle `ESMF_ArrayBundle` to be manipulated.

arrayList List of `ESMF_Array` objects to be added or used as replacement.

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

27.5.6 ESMF_ArrayBundleCreate - Create an ArrayBundle from a list of Arrays

INTERFACE:

```
  function ESMF_ArrayBundleCreate(arrayList, multiflag, &
    relaxedflag, name, rc)
```

RETURN VALUE:

```
    type(ESMF_ArrayBundle) :: ESMF_ArrayBundleCreate
```

ARGUMENTS:

```
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_Array), intent(in),  optional :: arrayList(:)
    logical,          intent(in),  optional :: multiflag
    logical,          intent(in),  optional :: relaxedflag
    character(len=*), intent(in),  optional :: name
    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:

Create an ESMF_ArrayBundle object from a list of existing Arrays.

The creation of an ArrayBundle leaves the bundled Arrays unchanged, they remain valid individual objects. An ArrayBundle is a light weight container of Array references. The actual data remains in place, there are no data movements or duplications associated with the creation of an ArrayBundle.

[arrayList] List of ESMF_Array objects to be bundled.

[multiflag] A setting of .true. allows multiple items with the same name to be added to arraybundle. For .false. added items must have unique names. The default setting is .false..

[relaxedflag] A setting of .true. indicates a relaxed definition of "add" under multiflag=.false. mode, where it is *not* an error if arrayList contains items with names that are also found in arraybundle. The arraybundle is left unchanged for these items. For .false. this is treated as an error condition. The default setting is .false..

[name] Name of the created ESMF_ArrayBundle. A default name is generated if not specified.

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

27.5.7 ESMF_ArrayBundleDestroy - Release resources associated with an ArrayBundle

INTERFACE:

```
subroutine ESMF_ArrayBundleDestroy(arraybundle, noGarbage, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle), intent(inout)           :: arraybundle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),   optional  :: noGarbage
     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 `noGarbage`. The argument provides a mechanism to override the default garbage collection mechanism when destroying an ESMF object.

DESCRIPTION:

Destroys an `ESMF_ArrayBundle` object. The member Arrays are not touched by this operation and remain valid objects that need to be destroyed individually if necessary.

By default a small remnant of the object is kept in memory in order to prevent problems with dangling aliases. The default garbage collection mechanism can be overridden with the `noGarbage` argument.

The arguments are:

arraybundle `ESMF_ArrayBundle` object to be destroyed.

[noGarbage] If set to `.TRUE.` the object will be fully destroyed and removed from the ESMF garbage collection system. Note however that under this condition ESMF cannot protect against accessing the destroyed object through dangling aliases – a situation which may lead to hard to debug application crashes.

It is generally recommended to leave the `noGarbage` argument set to `.FALSE.` (the default), and to take advantage of the ESMF garbage collection system which will prevent problems with dangling aliases or incorrect sequences of destroy calls. However this level of support requires that a small remnant of the object is kept in memory past the destroy call. This can lead to an unexpected increase in memory consumption over the course of execution in applications that use temporary ESMF objects. For situations where the repeated creation and destruction of temporary objects leads to memory issues, it is recommended to call with `noGarbage` set to `.TRUE.`, fully removing the entire temporary object from memory.

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

27.5.8 ESMF_ArrayBundleGet - Get object-wide information from an ArrayBundle

INTERFACE:

```
! Private name; call using ESMF_ArrayBundleGet()
subroutine ESMF_ArrayBundleGetListAll(arraybundle, &
  itemorderflag, arrayCount, arrayList, arrayNameList, name, rc)
```

ARGUMENTS:

```
type(ESMF_ArrayBundle),    intent(in)            :: arraybundle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_ItemOrder_Flag), intent(in),  optional :: itemorderflag
integer,                   intent(out), optional :: arrayCount
type(ESMF_Array),          intent(out), optional :: arrayList(:)
character(len=*),          intent(out), optional :: arrayNameList(:)
character(len=*),          intent(out), optional :: name
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:

  6.1.0 Added argument `itemorderflag`. The new argument gives the user control over the order in which the items are returned.

DESCRIPTION:

Get general, i.e. not Array name specific information from the ArrayBundle.

arraybundle `ESMF_ArrayBundle` to be queried.

[itemorderflag] Specifies the order of the returned items in the `arrayList` and `arrayNameList`. The default is `ESMF_ITEMORDER_ABC`. See **??** for a full list of options.

[arrayCount] Upon return holds the number of Arrays bundled in the ArrayBundle.

[arrayList] Upon return holds a list of Arrays bundled in `arraybundle`. The argument must be allocated to be at least of size `arrayCount`.

[arrayNameList] Upon return holds a list of the names of the Arrays bundled in `arraybundle`. The argument must be allocated to be at least of size `arrayCount`.

[name] Name of the ArrayBundle object.

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

27.5.9 ESMF_ArrayBundleGet - Get information about an Array by name and optionally return an Array

INTERFACE:

```
! Private name; call using ESMF_ArrayBundleGet()
subroutine ESMF_ArrayBundleGetItem(arraybundle, arrayName, &
  array, arrayCount, isPresent, rc)
```

ARGUMENTS:

```
type(ESMF_ArrayBundle), intent(in)            :: arraybundle
character(len=*),       intent(in)            :: arrayName
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_Array),       intent(out), optional :: array
integer,                intent(out), optional :: arrayCount
logical,                intent(out), optional :: isPresent
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:

Get information about items that match `arrayName` in ArrayBundle.

arraybundle `ESMF_ArrayBundle` to be queried.

arrayName Specified name.

[array] Upon return holds the requested Array item. It is an error if this argument was specified and there is not exactly one Array item in `arraybundle` that matches `arrayName`.

[arrayCount] Number of Arrays with `arrayName` in `arraybundle`.

[isPresent] Upon return indicates whether Array(s) with `arrayName` exist in `arraybundle`.

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

27.5.10 ESMF_ArrayBundleGet - Get a list of Arrays by name

INTERFACE:

```
! Private name; call using ESMF_ArrayBundleGet()
subroutine ESMF_ArrayBundleGetList(arraybundle, arrayName, arrayList, &
  itemorderflag, rc)
```

ARGUMENTS:

```
type(ESMF_ArrayBundle),    intent(in)            :: arraybundle
character(len=*),          intent(in)            :: arrayName
type(ESMF_Array),          intent(out)           :: arrayList(:)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_ItemOrder_Flag), intent(in),  optional :: itemorderflag
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:

  6.1.0 Added argument `itemorderflag`. The new argument gives the user control over the order in which the items are returned.

DESCRIPTION:

Get the list of Arrays from ArrayBundle that match `arrayName`.

arraybundle `ESMF_ArrayBundle` to be queried.

arrayName Specified name.

arrayList List of Arrays in `arraybundle` that match `arrayName`. The argument must be allocated to be at least of size `arrayCount` returned for this `arrayName`.

[itemorderflag] Specifies the order of the returned items in the `arrayList`. The default is `ESMF_ITEMORDER_ABC`. See **??** for a full list of options.

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

27.5.11 ESMF_ArrayBundleHalo - Execute an ArrayBundle halo operation

INTERFACE:

```
   subroutine ESMF_ArrayBundleHalo(arraybundle, routehandle, &
     checkflag, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle), intent(inout)          :: arraybundle
     type(ESMF_RouteHandle), intent(inout)          :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),   optional :: checkflag
     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:

Execute a precomputed ArrayBundle halo operation for the Arrays in `arrayBundle`.

See `ESMF_ArrayBundleHaloStore()` on how to precompute `routehandle`.

This call is *collective* across the current VM.

arraybundle `ESMF_ArrayBundle` containing data to be haloed.

routehandle Handle to the precomputed Route.

[checkflag] If set to `.TRUE.` the input Array pairs will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

27.5.12 ESMF_ArrayBundleHaloRelease - Release resources associated with an ArrayBundle halo operation

INTERFACE:

```
subroutine ESMF_ArrayBundleHaloRelease(routehandle, &
  noGarbage, rc)
```

ARGUMENTS:

```
  type(ESMF_RouteHandle), intent(inout)          :: routehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  logical,                intent(in),   optional :: noGarbage
  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:

  8.0.0 Added argument noGarbage. The argument provides a mechanism to override the default garbage collection mechanism when destroying an ESMF object.

DESCRIPTION:

Release resources associated with an ArrayBundle halo operation. After this call routehandle becomes invalid.

routehandle Handle to the precomputed Route.

[noGarbage] If set to .TRUE. the object will be fully destroyed and removed from the ESMF garbage collection system. Note however that under this condition ESMF cannot protect against accessing the destroyed object through dangling aliases – a situation which may lead to hard to debug application crashes.

It is generally recommended to leave the noGarbage argument set to .FALSE. (the default), and to take advantage of the ESMF garbage collection system which will prevent problems with dangling aliases or incorrect sequences of destroy calls. However this level of support requires that a small remnant of the object is kept in memory past the destroy call. This can lead to an unexpected increase in memory consumption over the course of execution in applications that use temporary ESMF objects. For situations where the repeated creation and destruction of temporary objects leads to memory issues, it is recommended to call with noGarbage set to .TRUE., fully removing the entire temporary object from memory.

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

27.5.13 ESMF_ArrayBundleHaloStore - Precompute an ArrayBundle halo operation

INTERFACE:

```
     subroutine ESMF_ArrayBundleHaloStore(arraybundle, routehandle, &
       startregion, haloLDepth, haloUDepth, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle),     intent(inout)          :: arraybundle
     type(ESMF_RouteHandle),     intent(inout)          :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_StartRegion_Flag),intent(in),  optional :: startregion
     integer,                    intent(in),  optional :: haloLDepth(:)
     integer,                    intent(in),  optional :: haloUDepth(:)
     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:

Store an ArrayBundle halo operation over the data in `arraybundle`. By default, i.e. without specifying `startregion`, `haloLDepth` and `haloUDepth`, all elements in the total Array regions that lie outside the exclusive regions will be considered potential destination elements for the halo operation. However, only those elements that have a corresponding halo source element, i.e. an exclusive element on one of the DEs, will be updated under the halo operation. Elements that have no associated source remain unchanged under halo.

Specifying `startregion` allows to change the shape of the effective halo region from the inside. Setting this flag to `ESMF_STARTREGION_COMPUTATIONAL` means that only elements outside the computational region for each Array are considered for potential destination elements for the halo operation. The default is `ESMF_STARTREGION_EXCLUSIVE`.

The `haloLDepth` and `haloUDepth` arguments allow to reduce the extent of the effective halo region. Starting at the region specified by `startregion`, the `haloLDepth` and `haloUDepth` define a halo depth in each direction. Note that the maximum halo region is limited by the total region for each Array, independent of the actual `haloLDepth` and `haloUDepth` setting. The total Array regions are local DE specific. The `haloLDepth` and `haloUDepth` are interpreted as the maximum desired extent, reducing the potentially larger region available for the halo operation.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_ArrayBundleHalo()` on any pair of ArrayBundles that matches `srcArrayBundle` and `dstArrayBundle` in *type*, *kind*, and memory layout of the *distributed* dimensions. However, the size, number, and index order of *undistributed* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This call is *collective* across the current VM.

arraybundle `ESMF_ArrayBundle` containing data to be haloed. The data in the halo regions may be destroyed by this call.

routehandle Handle to the precomputed Route.

[startregion] The start of the effective halo region on every DE. The default setting is `ESMF_STARTREGION_EXCLUSIVE`, rendering all non-exclusive elements potential halo destination elements. See section **??** for a complete list of valid settings.

[haloLDepth] This vector specifies the lower corner of the effective halo region with respect to the lower corner of `startregion`. The size of `haloLDepth` must equal the number of distributed Array dimensions.

[haloUDepth] This vector specifies the upper corner of the effective halo region with respect to the upper corner of `startregion`. The size of `haloUDepth` must equal the number of distributed Array dimensions.

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

27.5.14 ESMF_ArrayBundleIsCreated - Check whether an ArrayBundle object has been created

INTERFACE:

```
function ESMF_ArrayBundleIsCreated(arraybundle, rc)
```

RETURN VALUE:

```
logical :: ESMF_ArrayBundleIsCreated
```

ARGUMENTS:

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

DESCRIPTION:

Return `.true.` if the `arraybundle` has been created. Otherwise return `.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:

arraybundle `ESMF_ArrayBundle` queried.

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

27.5.15 ESMF_ArrayBundlePrint - Print ArrayBundle information

INTERFACE:

```
subroutine ESMF_ArrayBundlePrint(arraybundle, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle), intent(in)            :: arraybundle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     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:

Print internal information of the specified `ESMF_ArrayBundle` object to `stdout`.

The arguments are:

arraybundle `ESMF_ArrayBundle` object.

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

27.5.16 ESMF_ArrayBundleRead - Read Arrays to an ArrayBundle from file(s)

INTERFACE:

```
   subroutine ESMF_ArrayBundleRead(arraybundle, fileName, &
     singleFile, timeslice, iofmt, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle), intent(inout)          :: arraybundle
     character(*),           intent(in)             :: fileName
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),  optional  :: singleFile
     integer,                intent(in),  optional  :: timeslice
     type(ESMF_IOFmt_Flag),  intent(in),  optional  :: iofmt
     integer,                intent(out), optional  :: rc
```

DESCRIPTION:

Read Array data to an ArrayBundle object from file(s). For this API to be functional, the environment variable `ESMF_PIO` should be set to "internal" when the ESMF library is built. Please see the section on Data I/O, **??**.

Limitations:

- Only single tile Arrays are supported.
- Not supported in `ESMF_COMM=mpiuni` mode.

The arguments are:

arraybundle An `ESMF_ArrayBundle` object.

fileName The name of the file from which ArrayBundle data is read.

[singleFile] A logical flag, the default is .true., i.e., all Arrays in the bundle are stored in one single file. If .false., each Array is stored in separate files; these files are numbered with the name based on the argument "file". That is, a set of files are named: [file_name]001, [file_name]002, [file_name]003,...

[timeslice] The time-slice number of the variable read from file.

[iofmt] The I/O format. Please see Section **??** for the list of options. If not present, file names with a `.bin` extension will use `ESMF_IOFMT_BIN`, and file names with a `.nc` extension will use `ESMF_IOFMT_NETCDF`. Other files default to `ESMF_IOFMT_NETCDF`.

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

27.5.17 ESMF_ArrayBundleRedist - Execute an ArrayBundle redistribution

INTERFACE:

```
   subroutine ESMF_ArrayBundleRedist(srcArrayBundle, dstArrayBundle, &
     routehandle, checkflag, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle), intent(in),    optional :: srcArrayBundle
     type(ESMF_ArrayBundle), intent(inout), optional :: dstArrayBundle
     type(ESMF_RouteHandle), intent(inout)           :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),    optional :: checkflag
     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:

Execute a precomputed ArrayBundle redistribution from the Arrays in `srcArrayBundle` to the Arrays in `dstArrayBundle`.

The `srcArrayBundle` and `dstArrayBundle` arguments are optional in support of the situation where `srcArrayBundle` and/or `dstArrayBundle` are not defined on all PETs. The `srcArrayBundle` and `dstArrayBundle` must be specified on those PETs that hold source or destination DEs, respectively, but may be omitted on all other PETs. PETs that hold neither source nor destination DEs may omit both arguments.

This call is *collective* across the current VM.

[srcArrayBundle] `ESMF_ArrayBundle` with source data.

[dstArrayBundle] `ESMF_ArrayBundle` with destination data.

routehandle Handle to the precomputed Route.

[checkflag] If set to `.TRUE.` the input Array pairs will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

27.5.18 ESMF_ArrayBundleRedistRelease - Release resources associated with ArrayBundle redistribution

INTERFACE:

```
   subroutine ESMF_ArrayBundleRedistRelease(routehandle, &
     noGarbage, rc)
```

ARGUMENTS:

```
     type(ESMF_RouteHandle), intent(inout)          :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),   optional :: noGarbage
     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:

  8.0.0 Added argument `noGarbage`. The argument provides a mechanism to override the default garbage collection mechanism when destroying an ESMF object.

DESCRIPTION:

Release resources associated with an ArrayBundle redistribution. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

[noGarbage] If set to `.TRUE.` the object will be fully destroyed and removed from the ESMF garbage collection system. Note however that under this condition ESMF cannot protect against accessing the destroyed object through dangling aliases – a situation which may lead to hard to debug application crashes.

It is generally recommended to leave the noGarbage argument set to .FALSE. (the default), and to take advantage of the ESMF garbage collection system which will prevent problems with dangling aliases or incorrect sequences of destroy calls. However this level of support requires that a small remnant of the object is kept in memory past the destroy call. This can lead to an unexpected increase in memory consumption over the course of execution in applications that use temporary ESMF objects. For situations where the repeated creation and destruction of temporary objects leads to memory issues, it is recommended to call with noGarbage set to .TRUE., fully removing the entire temporary object from memory.

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

27.5.19 ESMF_ArrayBundleRedistStore - Precompute an ArrayBundle redistribution with local factor argument

INTERFACE:

```
! Private name; call using ESMF_ArrayBundleRedistStore()
subroutine ESMF_ArrayBundleRedistStore<type><kind>(srcArrayBundle, &
  dstArrayBundle, routehandle, factor, ignoreUnmatchedIndicesFlag, &
  srcToDstTransposeMap, rc)
```

ARGUMENTS:

```
  type(ESMF_ArrayBundle),  intent(in)            :: srcArrayBundle
  type(ESMF_ArrayBundle),  intent(inout)         :: dstArrayBundle
  type(ESMF_RouteHandle),  intent(inout)         :: routehandle
  <type>(ESMF_KIND_<kind>),intent(in)            :: factor
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  logical,                 intent(in),  optional :: ignoreUnmatchedIndicesFlag(:)
  integer,                 intent(in),  optional :: srcToDstTransposeMap(:)
  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:

  8.1.0 Added argument ignoreUnmatchedIndicesFlag to support cases where source and destination side do not cover the exact same index space.

DESCRIPTION:

Store an ArrayBundle redistribution operation from srcArrayBundle to dstArrayBundle. The redistribution between ArrayBundles is defined as the sequence of individual Array redistributions over all source and destination Array pairs in sequence. The method requires that srcArrayBundle and dstArrayBundle reference an identical number of ESMF_Array objects.

The effect of this method on ArrayBundles that contain aliased members is undefined.

PETs that specify a `factor` argument must use the <type><kind> overloaded interface. Other PETs call into the interface without `factor` argument. If multiple PETs specify the `factor` argument its type and kind as well as its value must match across all PETs. If none of the PETs specifies a `factor` argument the default will be a factor of 1.

See the description of method `ESMF_ArrayRedistStore()` for the definition of the Array based operation.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_ArrayBundleRedist()` on any pair of ArrayBundles that matches `srcArrayBundle` and `dstArrayBundle` in *type*, *kind*, and memory layout of the *distributed* dimensions. However, the size, number, and index order of *undistributed* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This method is overloaded for:
`ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`,
`ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

This call is *collective* across the current VM.

srcArrayBundle `ESMF_ArrayBundle` with source data.

dstArrayBundle `ESMF_ArrayBundle` with destination data. The data in these Arrays may be destroyed by this call.

routehandle Handle to the precomputed Route.

factor Factor by which to multiply source data.

[ignoreUnmatchedIndicesFlag] If set to .false., the *default*, source and destination side must cover the identical index space, using precisely matching sequence indices. If set to .true., mismatching sequence indices between source and destination side are silently ignored. The size of this array argument must either be 1 or equal the number of Arrays in the `srcArrayBundle` and `dstArrayBundle` arguments. In the latter case, the handling of unmatched indices is specified for each Array pair separately. If only one element is specified, it is used for *all* Array pairs.

[srcToDstTransposeMap] List with as many entries as there are dimensions in the Arrays in `srcArrayBundle`. Each entry maps the corresponding source Array dimension against the specified destination Array dimension. Mixing of distributed and undistributed dimensions is supported.

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

27.5.20 ESMF_ArrayBundleRedistStore - Precompute an ArrayBundle redistribution without local factor argument

INTERFACE:

```
! Private name; call using ESMF_ArrayBundleRedistStore()
subroutine ESMF_ArrayBundleRedistStoreNF(srcArrayBundle, dstArrayBundle, &
  routehandle, ignoreUnmatchedIndicesFlag, &
  srcToDstTransposeMap, rc)
```

ARGUMENTS:

```
    type(ESMF_ArrayBundle), intent(in)            :: srcArrayBundle
    type(ESMF_ArrayBundle), intent(inout)         :: dstArrayBundle
    type(ESMF_RouteHandle), intent(inout)         :: routehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    logical,                intent(in),  optional :: ignoreUnmatchedIndicesFlag(:)
    integer,                intent(in),  optional :: srcToDstTransposeMap(:)
    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:

  8.1.0 Added argument `ignoreUnmatchedIndicesFlag` to support cases where source and destination side do not cover the exact same index space.

DESCRIPTION:

Store an ArrayBundle redistribution operation from `srcArrayBundle` to `dstArrayBundle`. The redistribution between ArrayBundles is defined as the sequence of individual Array redistributions over all source and destination Array pairs in sequence. The method requires that `srcArrayBundle` and `dstArrayBundle` reference an identical number of `ESMF_Array` objects.

The effect of this method on ArrayBundles that contain aliased members is undefined.

PETs that specify a `factor` argument must use the <type><kind> overloaded interface. Other PETs call into the interface without `factor` argument. If multiple PETs specify the `factor` argument its type and kind as well as its value must match across all PETs. If none of the PETs specifies a `factor` argument the default will be a factor of 1.

See the description of method `ESMF_ArrayRedistStore()` for the definition of the Array based operation.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_ArrayBundleRedist()` on any pair of ArrayBundles that matches `srcArrayBundle` and `dstArrayBundle` in *type*, *kind*, and memory layout of the *distributed* dimensions. However, the size, number, and index order of *undistributed* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This call is *collective* across the current VM.

srcArrayBundle `ESMF_ArrayBundle` with source data.

dstArrayBundle `ESMF_ArrayBundle` with destination data. The data in these Arrays may be destroyed by this call.

routehandle Handle to the precomputed Route.

[ignoreUnmatchedIndicesFlag] If set to .false., the *default*, source and destination side must cover the identical index space, using precisely matching sequence indices. If set to .true., mismatching sequence indices between source and destination side are silently ignored. The size of this array argument must either be 1 or equal the number of Arrays in the `srcArrayBundle` and `dstArrayBundle` arguments. In the latter case, the handling of unmatched indices is specified for each Array pair separately. If only one element is specified, it is used for *all* Array pairs.

[srcToDstTransposeMap] List with as many entries as there are dimensions in the Arrays in `srcArrayBundle`. Each entry maps the corresponding source Array dimension against the specified destination Array dimension. Mixing of distributed and undistributed dimensions is supported.

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

27.5.21 ESMF_ArrayBundleRemove - Remove Arrays from ArrayBundle

INTERFACE:

```
subroutine ESMF_ArrayBundleRemove(arraybundle, arrayNameList, &
  multiflag, relaxedflag, rc)
```

ARGUMENTS:

```
type(ESMF_ArrayBundle), intent(inout)         :: arraybundle
character(len=*),       intent(in)            :: arrayNameList(:)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
logical,                intent(in),  optional :: multiflag
logical,                intent(in),  optional :: relaxedflag
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:

Remove Array(s) by name from ArrayBundle. In the relaxed setting it is *not* an error if `arrayNameList` contains names that are not found in `arraybundle`.

arraybundle `ESMF_ArrayBundle` from which to remove items.

arrayNameList List of items to remove.

[multiflag] A setting of `.true.` allows multiple Arrays with the same name to be removed from `arraybundle`. For `.false.`, items to be removed must have unique names. The default setting is `.false.`.

[relaxedflag] A setting of `.true.` indicates a relaxed definition of "remove" where it is *not* an error if `arrayNameList` contains item names that are not found in `arraybundle`. For `.false.` this is treated as an error condition. Further, in `multiflag=.false.` mode, the relaxed definition of "remove" also covers the case where there are multiple items in `arraybundle` that match a single entry in `arrayNameList`. For `relaxedflag=.false.` this is treated as an error condition. The default setting is `.false.`.

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

27.5.22 ESMF_ArrayBundleReplace - Replace Arrays in ArrayBundle

INTERFACE:

```
     subroutine ESMF_ArrayBundleReplace(arraybundle, arrayList, &
       multiflag, relaxedflag, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle), intent(inout)         :: arraybundle
     type(ESMF_Array),       intent(in)            :: arrayList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),  optional :: multiflag
     logical,                intent(in),  optional :: relaxedflag
     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:

Replace Array(s) by name in ArrayBundle. In the relaxed setting it is not an error if `arrayList` contains Arrays that do not match by name any item in `arraybundle`. These Arrays are simply ignored in this case.

arraybundle `ESMF_ArrayBundle` in which to replace items.

arrayList List of items to replace.

[multiflag] A setting of `.true.` allows multiple items with the same name to be replaced in `arraybundle`. For `.false.`, items to be replaced must have unique names. The default setting is `.false.`.

[relaxedflag] A setting of `.true.` indicates a relaxed definition of "replace" where it is *not* an error if `arrayList` contains items with names that are not found in `arraybundle`. These items in `arrayList` are ignored in the relaxed mode. For `.false.` this is treated as an error condition. Further, in `multiflag=.false.` mode, the relaxed definition of "replace" also covers the case where there are multiple items in `arraybundle` that match a single entry by name in `arrayList`. For `relaxedflag=.false.` this is treated as an error condition. The default setting is `.false.`.

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

27.5.23 ESMF_ArrayBundleSMM - Execute an ArrayBundle sparse matrix multiplication

INTERFACE:

```
   subroutine ESMF_ArrayBundleSMM(srcArrayBundle, dstArrayBundle, &
     routehandle, &
     zeroregion, & ! DEPRECATED ARGUMENT
     zeroregionflag, termorderflag, checkflag, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle),    intent(in),         optional :: srcArrayBundle
     type(ESMF_ArrayBundle),    intent(inout),      optional :: dstArrayBundle
     type(ESMF_RouteHandle),    intent(inout)                :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_Region_Flag),    intent(in), optional :: zeroregion ! DEPRECATED ARGUMENT
     type(ESMF_Region_Flag),    intent(in), target, optional :: zeroregionflag(:)
     type(ESMF_TermOrder_Flag), intent(in), target, optional :: termorderflag(:)
     logical,                   intent(in),         optional :: checkflag
     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 `termorderflag`. The new argument gives the user control over the order in which the src terms are summed up.

  8.1.0 Added argument `zeroregionflag`, and deprecated `zeroregion`. The new argument allows greater flexibility in setting the zero region for individual ArrayBundle members.

DESCRIPTION:

Execute a precomputed ArrayBundle sparse matrix multiplication from the Arrays in `srcArrayBundle` to the Arrays in `dstArrayBundle`.

The `srcArrayBundle` and `dstArrayBundle` arguments are optional in support of the situation where `srcArrayBundle` and/or `dstArrayBundle` are not defined on all PETs. The `srcArrayBundle` and `dstArrayBundle` must be specified on those PETs that hold source or destination DEs, respectively, but may be omitted on all other PETs. PETs that hold neither source nor destination DEs may omit both arguments.

This call is *collective* across the current VM.

[srcArrayBundle] `ESMF_ArrayBundle` with source data.

[dstArrayBundle] `ESMF_ArrayBundle` with destination data.

routehandle Handle to the precomputed Route.

[zeroregion] If set to `ESMF_REGION_TOTAL` *(default)* the total regions of all DEs in all Arrays in `dstArrayBundle` will be initialized to zero before updating the elements with the results of the sparse matrix multiplication. If set to `ESMF_REGION_EMPTY` the elements in the Arrays in `dstArrayBundle` will not be modified prior to the sparse matrix multiplication and results will be added to the incoming element values. Setting `zeroregion` to `ESMF_REGION_SELECT` will only zero out those elements in the destination Arrays that will be updated by the sparse matrix multiplication. See section **??** for a complete list of valid settings.

[zeroregionflag] If set to `ESMF_REGION_TOTAL` *(default)* the total regions of all DEs in the destination Array will be initialized to zero before updating the elements with the results of the sparse matrix multiplication. If set to `ESMF_REGION_EMPTY` the elements in the destination Array will not be modified prior to the sparse matrix

multiplication and results will be added to the incoming element values. A setting of ESMF_REGION_SELECT will only zero out those elements in the destination Array that will be updated by the sparse matrix multiplication. See section **??** for a complete list of valid settings. The size of this array argument must either be 1 or equal the number of Arrays in the srcArrayBundle and dstArrayBundle arguments. In the latter case, the zero region for each Array SMM operation is indicated separately. If only one zero region element is specified, it is used for *all* Array pairs.

[termorderflag] Specifies the order of the source side terms in all of the destination sums. The termorderflag only affects the order of terms during the execution of the RouteHandle. See the **??** section for an in-depth discussion of *all* bit-for-bit reproducibility aspects related to route-based communication methods. See **??** for a full list of options. The size of this array argument must either be 1 or equal the number of Arrays in the srcArrayBundle and dstArrayBundle arguments. In the latter case, the term order for each Array SMM operation is indicated separately. If only one term order element is specified, it is used for *all* Array pairs. The default is (/ESMF_TERMORDER_FREE/), allowing maximum flexibility in the order of terms for optimum performance.

[checkflag] If set to .TRUE. the input Array pairs will be checked for consistency with the precomputed operation provided by routehandle. If set to .FALSE. *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set checkflag to .FALSE. to achieve highest performance.

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

27.5.24 ESMF_ArrayBundleSMMRelease - Release resources associated with ArrayBundle sparse matrix multiplication

INTERFACE:

```
subroutine ESMF_ArrayBundleSMMRelease(routehandle, &
  noGarbage, rc)
```

ARGUMENTS:

```
    type(ESMF_RouteHandle), intent(inout)          :: routehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    logical,                intent(in),   optional :: noGarbage
    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:

  8.0.0 Added argument noGarbage. The argument provides a mechanism to override the default garbage collection mechanism when destroying an ESMF object.

DESCRIPTION:

Release resources associated with an ArrayBundle sparse matrix multiplication. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

[noGarbage] If set to `.TRUE.` the object will be fully destroyed and removed from the ESMF garbage collection system. Note however that under this condition ESMF cannot protect against accessing the destroyed object through dangling aliases – a situation which may lead to hard to debug application crashes.

It is generally recommended to leave the `noGarbage` argument set to `.FALSE.` (the default), and to take advantage of the ESMF garbage collection system which will prevent problems with dangling aliases or incorrect sequences of destroy calls. However this level of support requires that a small remnant of the object is kept in memory past the destroy call. This can lead to an unexpected increase in memory consumption over the course of execution in applications that use temporary ESMF objects. For situations where the repeated creation and destruction of temporary objects leads to memory issues, it is recommended to call with `noGarbage` set to `.TRUE.`, fully removing the entire temporary object from memory.

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

27.5.25 ESMF_ArrayBundleSMMStore - Precompute an ArrayBundle sparse matrix multiplication with local factors

INTERFACE:

```
! Private name; call using ESMF_ArrayBundleSMMStore()
subroutine ESMF_ArrayBundleSMMStore<type><kind>(srcArrayBundle, &
  dstArrayBundle, routehandle, factorList, factorIndexList, &
  ignoreUnmatchedIndicesFlag, srcTermProcessing, rc)
```

ARGUMENTS:

```
  type(ESMF_ArrayBundle),           intent(in)    :: srcArrayBundle
  type(ESMF_ArrayBundle),           intent(inout) :: dstArrayBundle
  type(ESMF_RouteHandle),           intent(inout) :: routehandle
  <type>(ESMF_KIND_<kind>), target, intent(in)    :: factorList(:)
  integer,                          intent(in)    :: factorIndexList(:,:)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  logical,                intent(in),    optional :: ignoreUnmatchedIndicesFlag(:)
  integer,                intent(inout), optional :: srcTermProcessing(:)
  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:

  7.1.0r Added argument `srcTermProcessing`. The new argument gives the user access to the tuning parameter affecting the sparse matrix execution and bit-wise reproducibility.

  8.1.0 Added argument `ignoreUnmatchedIndicesFlag` to support cases where the sparse matrix includes terms with source or destination sequence indices not present in the source or destination array.

DESCRIPTION:

Store an ArrayBundle sparse matrix multiplication operation from `srcArrayBundle` to `dstArrayBundle`. The sparse matrix multiplication between ArrayBundles is defined as the sequence of individual Array sparse matrix multiplications over all source and destination Array pairs in sequence. The method requires that `srcArrayBundle` and `dstArrayBundle` reference an identical number of `ESMF_Array` objects.

The effect of this method on ArrayBundles that contain aliased members is undefined.

PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* `factorList` and `factorIndexList` arguments.

See the description of method `ESMF_ArraySMMStore()` for the definition of the Array based operation.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_ArrayBundleSMM()` on any pair of ArrayBundles that matches `srcArrayBundle` and `dstArrayBundle` in *type*, *kind*, and memory layout of the *distributed* dimensions. However, the size, number, and index order of *undistributed* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This method is overloaded for:
`ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`,
`ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

This call is *collective* across the current VM.

srcArrayBundle `ESMF_ArrayBundle` with source data.

dstArrayBundle `ESMF_ArrayBundle` with destination data. The data in these Arrays may be destroyed by this call.

routehandle Handle to the precomputed Route.

factorList List of non-zero coefficients.

factorIndexList Pairs of sequence indices for the factors stored in `factorList`.

The second dimension of `factorIndexList` steps through the list of pairs, i.e. `size(factorIndexList,2) == size(factorList)`. The first dimension of `factorIndexList` is either of size 2 or size 4.

In the *size 2 format* `factorIndexList(1,:)` specifies the sequence index of the source element in the source Array while `factorIndexList(2,:)` specifies the sequence index of the destination element in the destination Array. For this format to be a valid option source and destination Arrays must have matching number

of tensor elements (the product of the sizes of all Array tensor dimensions). Under this condition an identity matrix can be applied within the space of tensor elements for each sparse matrix factor.

The *size 4 format* is more general and does not require a matching tensor element count. Here the `factorIndexList(1,:)` specifies the sequence index while `factorIndexList(2,:)` specifies the tensor sequence index of the source element in the source Array. Further `factorIndexList(3,:)` specifies the sequence index and `factorIndexList(4,:)` specifies the tensor sequence index of the destination element in the destination Array.

See section 28.2.18 for details on the definition of Array *sequence indices* and *tensor sequence indices*.

[ignoreUnmatchedIndicesFlag] If set to .false., the *default*, source and destination side must cover all of the squence indices defined in the sparse matrix. An error will be returned if a sequence index in the sparse matrix does not match on either the source or destination side. If set to .true., mismatching sequence indices are silently ignored. The size of this array argument must either be 1 or equal the number of Arrays in the `srcArrayBundle` and `dstArrayBundle` arguments. In the latter case, the handling of unmatched indices is specified for each Array pair separately. If only one element is specified, it is used for *all* Array pairs.

[srcTermProcessing] Source term summing options for route handle creation. See `ESMF_ArraySMMStore` documentation for a full parameter description. Two forms may be provided. If a single element list is provided, this integer value is applied across all bundle members. Otherwise, the list must contain as many elements as there are bundle members. For the special case of accessing the auto-tuned parameter (providing a negative integer value), the list length must equal the bundle member count.

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

27.5.26 ESMF_ArrayBundleSMMStore - Precompute an ArrayBundle sparse matrix multiplication without local factors

INTERFACE:

```
! Private name; call using ESMF_ArrayBundleSMMStore()
subroutine ESMF_ArrayBundleSMMStoreNF(srcArrayBundle, dstArrayBundle, &
  routehandle, ignoreUnmatchedIndicesFlag, srcTermProcessing, rc)
```

ARGUMENTS:

```
  type(ESMF_ArrayBundle),  intent(in)              :: srcArrayBundle
  type(ESMF_ArrayBundle),  intent(inout)           :: dstArrayBundle
  type(ESMF_RouteHandle),  intent(inout)           :: routehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  logical,                 intent(in),    optional :: ignoreUnmatchedIndicesFlag(:)
  integer,                 intent(inout), optional :: srcTermProcessing(:)
  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:

  7.1.0r Added argument `srcTermProcessing`. The new argument gives the user access to the tuning parameter affecting the sparse matrix execution and bit-wise reproducibility.

  8.1.0 Added argument `ignoreUnmatchedIndicesFlag` to support cases where the sparse matrix includes terms with source or destination sequence indices not present in the source or destination array.

DESCRIPTION:

Store an ArrayBundle sparse matrix multiplication operation from `srcArrayBundle` to `dstArrayBundle`. The sparse matrix multiplication between ArrayBundles is defined as the sequence of individual Array sparse matrix multiplications over all source and destination Array pairs in sequence. The method requires that `srcArrayBundle` and `dstArrayBundle` reference an identical number of `ESMF_Array` objects.

The effect of this method on ArrayBundles that contain aliased members is undefined.

PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* `factorList` and `factorIndexList` arguments.

See the description of method `ESMF_ArraySMMStore()` for the definition of the Array based operation.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_ArrayBundleSMM()` on any pair of ArrayBundles that matches `srcArrayBundle` and `dstArrayBundle` in *type*, *kind*, and memory layout of the *distributed* dimensions. However, the size, number, and index order of *undistributed* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This call is *collective* across the current VM.

srcArrayBundle `ESMF_ArrayBundle` with source data.

dstArrayBundle `ESMF_ArrayBundle` with destination data. The data in these Arrays may be destroyed by this call.

routehandle Handle to the precomputed Route.

[ignoreUnmatchedIndicesFlag] If set to .false., the *default*, source and destination side must cover all of the squence indices defined in the sparse matrix. An error will be returned if a sequence index in the sparse matrix does not match on either the source or destination side. If set to .true., mismatching sequence indices are silently ignored. The size of this array argument must either be 1 or equal the number of Arrays in the `srcArrayBundle` and `dstArrayBundle` arguments. In the latter case, the handling of unmatched indices is specified for each Array pair separately. If only one element is specified, it is used for *all* Array pairs.

[srcTermProcessing] Source term summing options for route handle creation. See `ESMF_ArraySMMStore` documentation for a full parameter description. Two forms may be provided. If a single element list is provided, this integer value is applied across all bundle members. Otherwise, the list must contain as many elements as there are bundle members. For the special case of accessing the auto-tuned parameter (providing a negative integer value), the list length must equal the bundle member count.

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

27.5.27 ESMF_ArrayBundleWrite - Write the Arrays into a file

INTERFACE:

```
   subroutine ESMF_ArrayBundleWrite(arraybundle, fileName, &
     convention, purpose, singleFile, overwrite, status, timeslice, iofmt, rc)
```

ARGUMENTS:

```
     type(ESMF_ArrayBundle),     intent(in)              :: arraybundle
     character(*),               intent(in)              :: fileName
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     character(*),               intent(in),  optional  :: convention
     character(*),               intent(in),  optional  :: purpose
     logical,                    intent(in),  optional  :: singleFile
     logical,                    intent(in),  optional  :: overwrite
     type(ESMF_FileStatus_Flag), intent(in),  optional  :: status
     integer,                    intent(in),  optional  :: timeslice
     type(ESMF_IOFmt_Flag),      intent(in),  optional  :: iofmt
     integer,                    intent(out), optional  :: rc
```

DESCRIPTION:

Write the Arrays into a file. For this API to be functional, the environment variable `ESMF_PIO` should be set to "internal" when the ESMF library is built. Please see the section on Data I/O, **??**.

When `convention` and `purpose` arguments are specified, NetCDF dimension labels and variable attributes are written from each Array in the ArrayBundle from the corresponding Attribute package. Additionally, Attributes may be set on the ArrayBundle level under the same Attribute package. This allows the specification of global attributes within the file. As with individual Arrays, the value associated with each name may be either a scalar character string, or a scalar or array of type integer, real, or double precision.

Limitations:

- Only single tile Arrays are supported.
- Not supported in `ESMF_COMM=mpiuni` mode.

The arguments are:

arraybundle An `ESMF_ArrayBundle` object.

fileName The name of the output file to which array bundle data is written.

[convention] Specifies an Attribute package associated with the ArrayBundle, and the contained Arrays, used to create NetCDF dimension labels and attributes in the file. When this argument is present, the `purpose` argument must also be present. Use this argument only with a NetCDF I/O format. If binary format is used, ESMF will return an error code.

[purpose] Specifies an Attribute package associated with the ArrayBundle, and the contained Arrays, used to create NetCDF dimension labels and attributes in the file. When this argument is present, the `convention` argument must also be present. Use this argument only with a NetCDF I/O format. If binary format is used, ESMF will return an error code.

[singleFile] A logical flag, the default is .true., i.e., all arrays in the bundle are written in one single file. If .false., each array will be written in separate files; these files are numbered with the name based on the argument "file". That is, a set of files are named: [file_name]001, [file_name]002, [file_name]003,...

[overwrite] A logical flag, the default is .false., i.e., existing Array data may *not* be overwritten. If .true., the overwrite behavior depends on the value of `iofmt` as shown below:

`iofmt` = `ESMF_IOFMT_BIN`: All data in the file will be overwritten with each Arrays's data.

`iofmt` = `ESMF_IOFMT_NETCDF, ESMF_IOFMT_NETCDF_64BIT_OFFSET`: Only the data corresponding to each Array's name will be be overwritten. If the `timeslice` option is given, only data for the given timeslice may be overwritten. Note that it is always an error to attempt to overwrite a NetCDF variable with data which has a different shape.

[status] The file status. Please see Section **??** for the list of options. If not present, defaults to `ESMF_FILESTATUS_UNKNOWN`.

[timeslice] Some I/O formats (e.g. NetCDF) support the output of data in form of time slices. The `timeslice` argument provides access to this capability. `timeslice` must be positive. The behavior of this option may depend on the setting of the `overwrite` flag:

`overwrite = .false.`: If the timeslice value is less than the maximum time already in the file, the write will fail.

`overwrite = .true.`: Any positive timeslice value is valid.

By default, i.e. by omitting the `timeslice` argument, no provisions for time slicing are made in the output file, however, if the file already contains a time axis for the variable, a timeslice one greater than the maximum will be written.

[iofmt] The I/O format. Please see Section **??** for the list of options. If not present, file names with a `.bin` extension will use `ESMF_IOFMT_BIN`, and file names with a `.nc` extension will use `ESMF_IOFMT_NETCDF`. Other files default to `ESMF_IOFMT_NETCDF`.

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

28 Array Class

28.1 Description

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

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

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

system image (SSI). This aspect is managed by the ESMF DELayout class. Pinning to PET is the most common mode and is the default.

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

28.2 Use and Examples

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

28.2.1 Array from native Fortran array with 1 DE per PET

The create call of the `ESMF_Array` class has been overloaded extensively to facilitate the need for generality while keeping simple cases simple. The following program demonstrates one of the simpler cases, where existing local Fortran arrays are to be used to provide the PET-local memory allocations for the Array object.

```
program ESMF_ArrayFarrayEx


  use ESMF
  use ESMF_TestMod

  implicit none
```

The Fortran language provides a variety of ways to define and allocate an array. Actual Fortran array objects must either be explicit-shape or deferred-shape. In the first case the memory allocation and deallocation is automatic from the user's perspective and the details of the allocation (static or dynamic, heap or stack) are left to the compiler. (Compiler flags may be used to control some of the details). In the second case, i.e. for deferred-shape actual objects, the array definition must include the `pointer` or `allocatable` attribute and it is the user's responsibility to allocate memory. While it is also the user's responsibility to deallocate memory for arrays with the `pointer` attribute the compiler will automatically deallocate allocatable arrays under certain circumstances defined by the Fortran standard.

The `ESMF_ArrayCreate()` interface has been written to accept native Fortran arrays of any flavor as a means to allow user-controlled memory management. The Array create call will check on each PET if sufficient memory has been provided by the specified Fortran arrays and will indicate an error if a problem is detected. However, the Array create call cannot validate the lifetime of the provided memory allocations. If, for instance, an Array object was created in a subroutine from an automatic explicit-shape array or an allocatable array, the memory allocations referenced by the Array object will be automatically deallocated on return from the subroutine unless provisions are made by the application writer to prevent such behavior. The Array object cannot control when memory that has been provided by the user during Array creation becomes deallocated, however, the Array will indicate an error if its memory references have been invalidated.

The easiest, portable way to provide safe native Fortran memory allocations to Array create is to use arrays with the `pointer` attribute. Memory allocated for an array pointer will not be deallocated automatically. However, in this

case the possibility of memory leaks becomes an issue of concern. The deallocation of memory provided to an Array in form of a native Fortran allocation will remain the users responsibility.

None of the concerns discussed above are an issue in this example where the native Fortran array `farray` is defined in the main program. All different types of array memory allocation are demonstrated in this example. First `farrayE` is defined as a 2D explicit-shape array on each PET which will automatically provide memory for 10×10 elements.

```
! local variables
real(ESMF_KIND_R8)       :: farrayE(10,10)  ! explicit shape Fortran array
```

Then an allocatable array `farrayA` is declared which will be used to show user-controlled dynamic memory allocation.

```
real(ESMF_KIND_R8), allocatable :: farrayA(:,:) ! allocatable Fortran array
```

Finally an array with pointer attribute `farrayP` is declared, also used for user-controlled dynamic memory allocation.

```
real(ESMF_KIND_R8), pointer :: farrayP(:,:)   ! Fortran array pointer
```

A matching array pointer must also be available to gain access to the arrays held by an Array object.

```
real(ESMF_KIND_R8), pointer :: farrayPtr(:,:) ! matching Fortran array ptr
type(ESMF_DistGrid)         :: distgrid       ! DistGrid object
type(ESMF_Array)            :: array          ! Array object
integer                     :: rc



call ESMF_Initialize(defaultlogfilename="ArrayFarrayEx.Log", &
                     logkindflag=ESMF_LOGKIND_MULTI, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

On each PET `farrayE` can be accessed directly to initialize the entire PET-local array.

```
farrayE = 12.45d0 ! initialize to some value
```

In order to create an Array object a DistGrid must first be created that describes the total index space and how it is decomposed and distributed. In the simplest case only the `minIndex` and `maxIndex` of the total space must be provided.

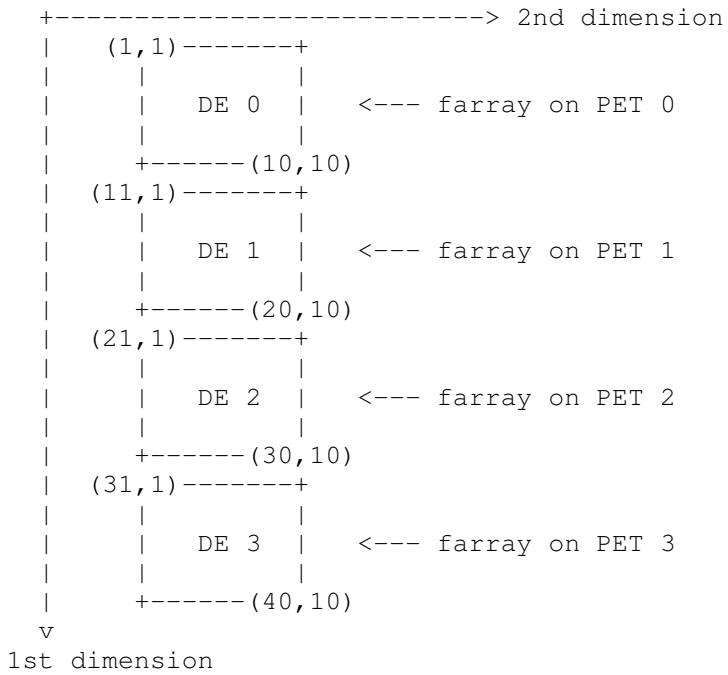
```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/40,10/), rc=rc)
```

This example is assumed to run on 4 PETs. The default 2D decomposition will then be into 4 x 1 DEs as to ensure 1 DE per PET.

Now the Array object can be created using the `farrayE` and the DistGrid just created.

```
array = ESMF_ArrayCreate(farray=farrayE, distgrid=distgrid, &
  indexflag=ESMF_INDEX_DELOCAL, rc=rc)
```

The 40 x 10 index space defined by the `minIndex` and `maxIndex` arguments paired with the default decomposition will result in the following distributed Array.



Providing `farrayE` during Array creation does not change anything about the actual `farrayE` object. This means that each PET can use its local `farrayE` directly to access the memory referenced by the Array object.

```
print *, farrayE
```

Another way of accessing the memory associated with an Array object is to use `ArrayGet()` to obtain an Fortran pointer that references the PET-local array.

```
call ESMF_ArrayGet(array, farrayPtr=farrayPtr, rc=rc)
```

```
print *, farrayPtr
```

Finally the Array object must be destroyed. The PET-local memory of the `farrayE`s will remain in user control and will not be altered by `ArrayDestroy()`.

```
  call ESMF_ArrayDestroy(array, rc=rc)
```

Since the memory allocation for each `farrayE` is automatic there is nothing more to do.

The interaction between `farrayE` and the Array class is representative also for the two other cases `farrayA` and `farrayP`. The only difference is in the handling of memory allocations.

```
  allocate(farrayA(10,10))    ! user controlled allocation
  farrayA = 23.67d0           ! initialize to some value
  array = ESMF_ArrayCreate(farray=farrayA, distgrid=distgrid, &
    indexflag=ESMF_INDEX_DELOCAL, rc=rc)


  print *, farrayA            ! print PET-local farrayA directly
  call ESMF_ArrayGet(array, farrayPtr=farrayPtr, rc=rc)! obtain array pointer
  print *, farrayPtr          ! print PET-local piece of Array through pointer
  call ESMF_ArrayDestroy(array, rc=rc) ! destroy the Array
  deallocate(farrayA)         ! user controlled de-allocation
```

The `farrayP` case is identical.

```
  allocate(farrayP(10,10))    ! user controlled allocation
  farrayP = 56.81d0           ! initialize to some value
  array = ESMF_ArrayCreate(farray=farrayP, distgrid=distgrid, &
    indexflag=ESMF_INDEX_DELOCAL, rc=rc)


  print *, farrayP            ! print PET-local farrayA directly
  call ESMF_ArrayGet(array, farrayPtr=farrayPtr, rc=rc)! obtain array pointer
  print *, farrayPtr          ! print PET-local piece of Array through pointer
  call ESMF_ArrayDestroy(array, rc=rc) ! destroy the Array
  deallocate(farrayP)         ! user controlled de-allocation
```

To wrap things up the DistGrid object is destroyed and ESMF can be finalized.

```
  call ESMF_DistGridDestroy(distgrid, rc=rc) ! destroy the DistGrid



  call ESMF_Finalize(rc=rc)


end program
```

28.2.2 Array from native Fortran array with extra elements for halo or padding

The example of the previous section showed how easy it is to create an Array object from existing PET-local Fortran arrays. The example did, however, not define any halo elements around the DE-local regions. The following code demonstrates how an Array object with space for a halo can be set up.

```
program ESMF_ArrayFarrayHaloEx


  use ESMF
  use ESMF_TestMod

  implicit none
```

The allocatable array `farrayA` will be used to provide the PET-local Fortran array for this example.

```
  ! local variables
  real(ESMF_KIND_R8), allocatable :: farrayA(:,:) ! allocatable Fortran array
  real(ESMF_KIND_R8), pointer :: farrayPtr(:,:)   ! matching Fortran array ptr
  type(ESMF_DistGrid)         :: distgrid         ! DistGrid object
  type(ESMF_Array)            :: array            ! Array object
  integer                     :: rc, i, j
  real(ESMF_KIND_R8)          :: localSum



  call ESMF_Initialize(defaultlogfilename="ArrayFarrayHaloEx.Log", &
                       logkindflag=ESMF_LOGKIND_MULTI, rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

The Array is to cover the exact same index space as in the previous example. Furthermore decomposition and distribution are also kept the same. Hence the same DistGrid object will be created and it is expected to execute this example with 4 PETs.

```
  distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/40,10/), rc=rc)
```

This DistGrid describes a 40 x 10 index space that will be decomposed into 4 DEs when executed on 4 PETs, associating 1 DE per PET. Each DE-local exclusive region contains 10 x 10 elements. The DistGrid also stores and provides information about the relationship between DEs in index space, however, DistGrid does not contain information about halos. Arrays contain halo information and it is possible to create multiple Arrays covering the same index space with identical decomposition and distribution using the same DistGrid object, while defining different, Array-specific halo regions.

The extra memory required to cover the halo in the Array object must be taken into account when allocating the PET-local `farrayA` arrays. For a halo of 2 elements in each direction the following allocation will suffice.

```
  allocate(farrayA(14,14))    ! Fortran array with halo: 14 = 10 + 2 * 2
```

The farrayA can now be used to create an Array object with enough space for a two element halo in each direction. The Array creation method checks for each PET that the local Fortran array can accommodate the requested regions.

The default behavior of ArrayCreate() is to center the exclusive region within the total region. Consequently the following call will provide the 2 extra elements on each side of the exclusive 10 x 10 region without having to specify any additional arguments.

```
array = ESMF_ArrayCreate(farray=farrayA, distgrid=distgrid, &
  indexflag=ESMF_INDEX_DELOCAL, rc=rc)
```

The exclusive Array region on each PET can be accessed through a suitable Fortran array pointer. See section 28.2.6 for more details on Array regions.

```
call ESMF_ArrayGet(array, farrayPtr=farrayPtr, rc=rc)
```

Following Array bounds convention, which by default puts the beginning of the exclusive region at (1, 1, ...), the following loop will add up the values of the local exclusive region for each DE, regardless of how the bounds were chosen for the original PET-local farrayA arrays.

```
localSum = 0.
do j=1, 10
  do i=1, 10
    localSum = localSum + farrayPtr(i, j)
  enddo
enddo
```

Elements with i or j in the [-1,0] or [11,12] ranges are located outside the exclusive region and may be used to define extra computational points or halo operations.

Cleanup and shut down ESMF.

```
call ESMF_ArrayDestroy(array, rc=rc)


deallocate(farrayA)
call ESMF_DistGridDestroy(distgrid, rc=rc)


call ESMF_Finalize(rc=rc)


end program
```

28.2.3 Array from ESMF_LocalArray

Alternative to the direct usage of Fortran arrays during Array creation it is also possible to first create an ESMF_LocalArray and create the Array from it. While this may seem more burdensome for the 1 DE per PET

cases discussed in the previous sections it allows a straightforward generalization to the multiple DE per PET case. The following example first recaptures the previous example using an ESMF_LocalArray and then expands to the multiple DE per PET case.

```
program ESMF_ArrayLarrayEx


  use ESMF
  use ESMF_TestMod

  implicit none
```

The current ESMF_LocalArray interface requires Fortran arrays to be defined with pointer attribute.

```
  ! local variables
  real(ESMF_KIND_R8), pointer :: farrayP(:,:)   ! Fortran array pointer
  real(ESMF_KIND_R8), pointer :: farrayPtr(:,:) ! matching Fortran array ptr
  type(ESMF_LocalArray)       :: larray         ! ESMF_LocalArray object
  type(ESMF_LocalArray)       :: larrayRef      ! ESMF_LocalArray object
  type(ESMF_DistGrid)         :: distgrid       ! DistGrid object
  type(ESMF_Array)            :: array          ! Array object
  integer                     :: rc, i, j, de
  real(ESMF_KIND_R8)          :: localSum
  type(ESMF_LocalArray), allocatable :: larrayList(:) ! LocalArray object list
  type(ESMF_LocalArray), allocatable :: larrayRefList(:)!LocalArray obj. list

  type(ESMF_VM):: vm
  integer:: localPet, petCount



  call ESMF_Initialize(vm=vm, defaultlogfilename="ArrayLarrayEx.Log", &
                    logkindflag=ESMF_LOGKIND_MULTI, rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  call ESMF_VMGet(vm, localPet=localPet, petCount=petCount, rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)

  if (petCount /= 4) then
    finalrc = ESMF_FAILURE
    goto 10
  endif
```

DistGrid and array allocation remains unchanged.

```
  distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/40,10/), rc=rc)


  allocate(farrayP(14,14))    ! allocate Fortran array on each PET with halo
```

Now instead of directly creating an Array object using the PET-local `farrayP`s an `ESMF_LocalArray` object will be created on each PET.

```
larray = ESMF_LocalArrayCreate(farrayP, &
             datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)
```

The Array object can now be created from `larray`. The Array creation method checks for each PET that the LocalArray can accommodate the requested regions.

```
array = ESMF_ArrayCreate(localarrayList=(/larray/), distgrid=distgrid, rc=rc)
```

Once created there is no difference in how the Array object can be used. The exclusive Array region on each PET can be accessed through a suitable Fortran array pointer as before.

```
call ESMF_ArrayGet(array, farrayPtr=farrayPtr, rc=rc)
```

Alternatively it is also possible (independent of how the Array object was created) to obtain the reference to the array allocation held by Array in form of an `ESMF_LocalArray` object. The `farrayPtr` can then be extracted using LocalArray methods.

```
call ESMF_ArrayGet(array, localarray=larrayRef, rc=rc)


call ESMF_LocalArrayGet(larrayRef, farrayPtr, rc=rc)
```

Either way the `farrayPtr` reference can be used now to add up the values of the local exclusive region for each DE. The following loop works regardless of how the bounds were chosen for the original PET-local `farrayP` arrays and consequently the PET-local `larray` objects.

```
localSum = 0.
do j=1, 10
  do i=1, 10
    localSum = localSum + farrayPtr(i, j)
  enddo
enddo
print *, "localSum=", localSum
```

Cleanup.

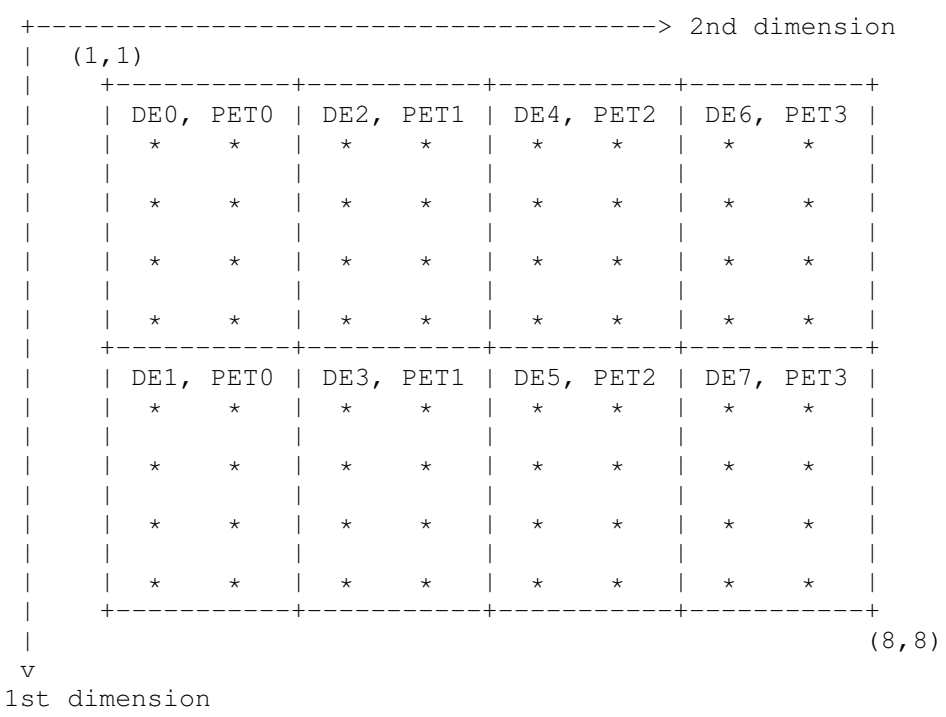
```
call ESMF_ArrayDestroy(array, rc=rc)
call ESMF_LocalArrayDestroy(larray, rc=rc)
deallocate(farrayP)   ! use the pointer that was used in allocate statement
call ESMF_DistGridDestroy(distgrid, rc=rc)
```

While the usage of LocalArrays is unnecessarily cumbersome for 1 DE per PET Arrays, it provides a straightforward path for extending the interfaces to multiple DEs per PET.

In the following example a 8 x 8 index space will be decomposed into 2 x 4 = 8 DEs. The situation is captured by the following DistGrid object.

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/8,8/), &
  regDecomp=(/2,4/), rc=rc)
```

The `distgrid` object created in this manner will contain 8 DEs no matter how many PETs are available during execution. Assuming an execution on 4 PETs will result in the following distribution of the decomposition.



Obviously each PET is associated with 2 DEs. Each PET must allocate enough space for *all* its DEs. This is done by allocating as many DE-local arrays as there are DEs on the PET. The reference to these array allocations is passed into ArrayCreate via a LocalArray list argument that holds as many elements as there are DEs on the PET. Here each PET must allocate for two DEs.

```
allocate(larrayList(2))   ! 2 DEs per PET
allocate(farrayP(4, 2))   ! without halo each DE is of size 4 x 2
farrayP = 123.456d0
larrayList(1) = ESMF_LocalArrayCreate(farrayP, &
  datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc) !1st DE
allocate(farrayP(4, 2))   ! without halo each DE is of size 4 x 2
farrayP = 456.789d0
larrayList(2) = ESMF_LocalArrayCreate(farrayP, &
```

```
    datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc) !2nd DE
```

Notice that it is perfectly fine to *re*-use `farrayP` for all allocations of DE-local Fortran arrays. The allocated memory can be deallocated at the end using the array pointer contained in the `larrayList`.

With this information an Array object can be created. The `distgrid` object indicates 2 DEs for each PET and ArrayCreate() expects to find two LocalArray elements in `larrayList`.

```
  array = ESMF_ArrayCreate(localarrayList=larrayList, distgrid=distgrid, rc=rc)
```

Usage of a LocalArray list is the only way to provide a list of variable length of Fortran array allocations to ArrayCreate() for each PET. The `array` object created by the above call is an ESMF distributed object. As such it must follow the ESMF convention that requires that the call to `ESMF_ArrayCreate()` must be issued in unison by all PETs of the current context. Each PET only calls ArrayCreate() once, even if there are multiple DEs per PET.

The ArrayGet() method provides access to the list of LocalArrays on each PET.

```
  allocate(larrayRefList(2))
  call ESMF_ArrayGet(array, localarrayList=larrayRefList, rc=rc)
```

Finally, access to the actual Fortran pointers is done on a per DE basis. Generally each PET will loop over its DEs.

```
  do de=1, 2
    call ESMF_LocalArrayGet(larrayRefList(de), farrayPtr, rc=rc)
    localSum = 0.
    do j=1, 2
      do i=1, 4
        localSum = localSum + farrayPtr(i, j)
      enddo
    enddo
    print *, "localSum=", localSum
  enddo
```

Note: If the VM associates multiple PEs with a PET the application writer may decide to use OpenMP loop parallelization on the `de` loop.

Cleanup requires that the PET-local deallocations are done before the pointers to the actual Fortran arrays are lost. Notice that `larrayList` is used to obtain the pointers used in the deallocate statement. Pointers obtained from the `larrayRefList`, while pointing to the same data, *cannot* be used to deallocate the array allocations!

```
  do de=1, 2
    call ESMF_LocalArrayGet(larrayList(de), farrayPtr, rc=rc)


    deallocate(farrayPtr)
    call ESMF_LocalArrayDestroy(larrayList(de), rc=rc)
```

```
  enddo
  deallocate(larrayList)
  deallocate(larrayRefList)
  call ESMF_ArrayDestroy(array, rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  call ESMF_DistGridDestroy(distgrid, rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

With that ESMF can be shut down cleanly.

```
  call ESMF_Finalize(rc=rc)


end program
```

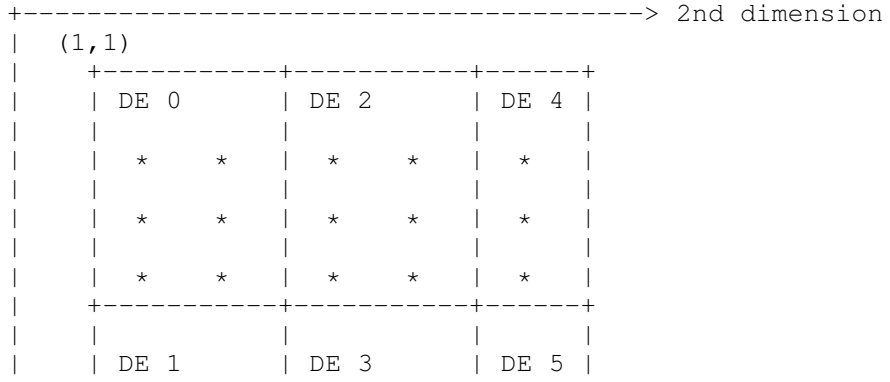
28.2.4 Create Array with automatic memory allocation

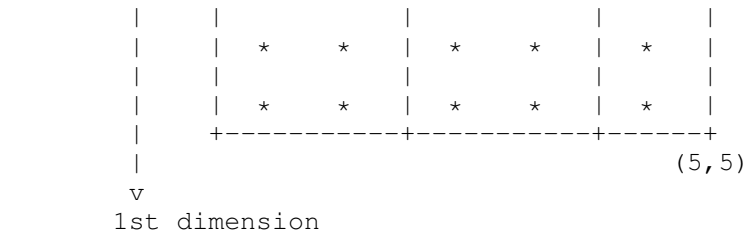
In the examples of the previous sections the user provided memory allocations for each of the DE-local regions for an Array object. The user was able to use any of the Fortran methods to allocate memory, or go through the `ESMF_LocalArray` interfaces to obtain memory allocations before passing them into ArrayCreate(). Alternatively ESMF offers methods that handle Array memory allocations inside the library.

As before, to create an `ESMF_Array` object an `ESMF_DistGrid` must be created. The DistGrid object holds information about the entire index space and how it is decomposed into DE-local exclusive regions. The following line of code creates a DistGrid for a 5x5 global index space that is decomposed into 2 x 3 = 6 DEs.

```
  distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
    regDecomp=(/2,3/), rc=rc)
```

The following is a representation of the index space and its decomposition into DEs. Each asterisk (*) represents a single element.





Besides the DistGrid it is the *type, kind* and *rank* information, "tkr" for short, that is required to create an Array object. It turns out that the rank of the Array object is fully determined by the DistGrid and other (optional) arguments passed into ArrayCreate(), so that explicit specification of the Array rank is redundant.

The simplest way to supply the type and kind information of the Array is directly through the `typekind` argument. Here a double precision Array is created on the previously created DistGrid. Since no other arguments are specified that could alter the rank of the Array it becomes equal to the dimCount of the DistGrid, i.e a 2D Array is created on top of the DistGrid.

```
array = ESMF_ArrayCreate(typekind=ESMF_TYPEKIND_R8, distgrid=distgrid, rc=rc)
```

The different methods on how an Array object is created have no effect on the use of `ESMF_ArrayDestroy()`.

```
call ESMF_ArrayDestroy(array, rc=rc)
```

Alternatively the same Array can be created specifying the "tkr" information in form of an ArraySpec variable. The ArraySpec explicitly contains the Array rank and thus results in an over specification on the ArrayCreate() interface. ESMF checks all input information for consistency and returns appropriate error codes in case any inconsistencies are found.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)


array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, rc=rc)
```

The Array object created by the above call is an ESMF distributed object. As such it must follow the ESMF convention that requires that the call to `ESMF_ArrayCreate()` must be issued in unison by all PETs of the current context.

28.2.5 Native language memory access

There are two different methods by which the user can access the data held inside an ESMF Array object. The first method provides direct access to a native language array object. Specifically, the `farrayPtr` argument returned by `ESMF_ArrayGet()` is a Fortran array pointer that can be used to access the PET-local data inside the Array object.

Many applications work in the 1 DE per PET mode, with exactly one DE on every PET. Accessing the Array memory on each PET for this situation is especially simple as is shown in section 28.2.1. However, the Array class is not restricted to the special 1 DE per PET case, but supports multiple separate memory allocations on each PET. The

number of such PET-local allocations is given by the `localDeCount`, i.e. there is one memory allocation for every DE that is associated with the local PET.

Access to a specific local memory allocation of an Array object is still accomplished by returning the `farrayPtr` argument. However, for $localDeCount > 1$ the formally optional `localDe` argument to `ESMF_ArrayGet()` turns into a practically required argument. While in general the `localDe` in ESMF is simply a local index variable that enumerates the DEs that are associated with the local PET (e.g. see section **??**), the bounds of this index variable are strictly defined as `[0,...,localDeCount-1]` when it is used as an input argument. The following code demonstrates this.

First query the Array for `localDeCount`. This number may be different on each PET and indicates how many DEs are mapped against the local PET.

```
call ESMF_ArrayGet(array, localDeCount=localDeCount, rc=rc)
```

Looping the `localDe` index variable from 0 to `localDeCount-1` allows access to each of the local memory allocations of the Array object:

```
do localDe=0, localDeCount-1
  call ESMF_ArrayGet(array, farrayPtr=myFarray, localDe=localDe, rc=rc)


  ! use myFarray to access local DE data
enddo
```

The second method to access the memory allocations in an Array object is to go through the ESMF LocalArray object. To this end the Array is queried for a list of PET-local LocalArray objects. The LocalArray objects in the list correspond to the DEs on the local PET. Here the `localDe` argument is solely a user level index variable, and in principle the lower bound can be chosen freely. However, for better alignment with the previous case (where `localDe` served as an input argument to an ESMF method) the following example again fixes the lower bound at zero.

```
allocate(larrayList(0:localDeCount-1))
call ESMF_ArrayGet(array, localarrayList=larrayList, rc=rc)


do localDe=0, localDeCount-1
  call ESMF_LocalArrayGet(larrayList(localDe), myFarray, &
     datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)


  ! use myFarray to access local DE data
enddo
```

See section 28.2.3 for more on LocalArray usage in Array. In most cases memory access through a LocalArray list is less convenient than the direct `farrayPtr` method because it adds an extra object level between the ESMF Array and the native language array.

28.2.6 Regions and default bounds

Each `ESMF_Array` object is decomposed into DEs as specified by the associated `ESMF_DistGrid` object. Each piece of this decomposition, i.e. each DE, holds a chunk of the Array data in its own local piece of memory. The details of the Array decomposition are described in the following paragraphs.

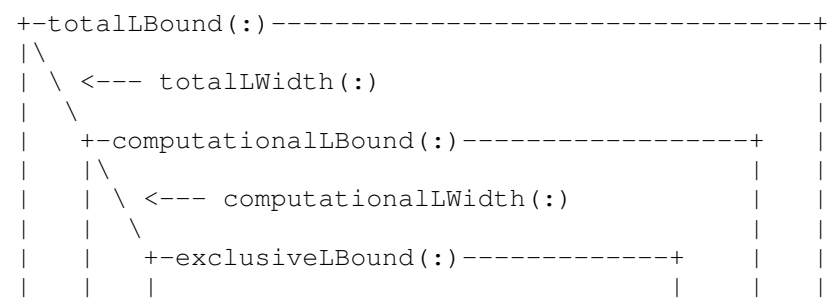
At the center of the Array decomposition is the `ESMF_DistGrid` class. The DistGrid object specified during Array creation contains three essential pieces of information:

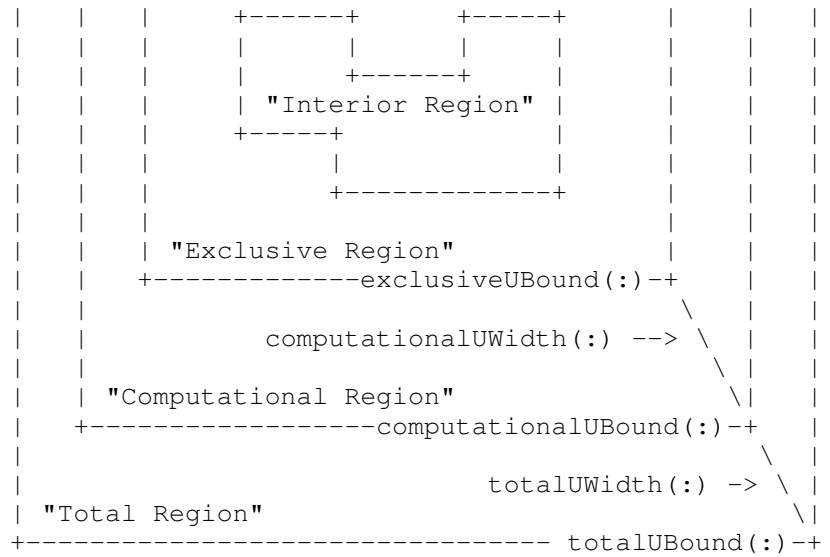
- The extent and topology of the global domain covered by the Array object in terms of indexed elements. The total extent may be a composition of smaller logically rectangular (LR) domain pieces called tiles.
- The decomposition of the entire domain into "element exclusive" DE-local LR chunks. *Element exclusive* means that there is no element overlap between DE-local chunks. This, however, does not exclude degeneracies on edge boundaries for certain topologies (e.g. bipolar).
- The layout of DEs over the available PETs and thus the distribution of the Array data.

Each element of an Array is associated with a *single* DE. The union of elements associated with a DE, as defined by the DistGrid above, corresponds to a LR chunk of index space, called the *exclusive region* of the DE.

There is a hierarchy of four regions that can be identified for each DE in an Array object. Their definition and relationship to each other is as follows:

- *Interior Region*: Region that only contains local elements that are *not* mapped into the halo of any other DE. The shape and size of this region for a particular DE depends non-locally on the halos defined by other DEs and may change during computation as halo operations are precomputed and released. Knowledge of the interior elements may be used to improve performance by overlapping communications with ongoing computation for a DE.
- *Exclusive Region*: Elements for which a DE claims exclusive ownership. Practically this means that the DE will be the sole source for these elements in halo and reduce operations. There are exceptions to this in some topologies. The exclusive region includes all elements of the interior region.
- *Computational Region*: Region that can be set arbitrarily within the bounds of the total region (defined next). The typical use of the computation region is to define bounds that only include elements that are updated by a DE-local computation kernel. The computational region does not need to include all exclusive elements and it may also contain elements that lie outside the exclusive region.
- *Total (Memory) Region*: Total of all DE-locally allocated elements. The size and shape of the total memory region must accommodate the union of exclusive and computational region but may contain additional elements. Elements outside the exclusive region may overlap with the exclusive region of another DE which makes them potential receivers for Array halo operations. Elements outside the exclusive region that do not overlap with the exclusive region of another DE can be used to set boundary conditions and/or serve as extra memory padding.





With the following definitions:

```
computationalLWidth(:) = exclusiveLBound(:) - computationalLBound(:)
computationalUWidth(:) = computationalUBound(:) - exclusiveUBound(:)
```

and

```
totalLWidth(:) = exclusiveLBound(:) - totalLBound(:)
totalUWidth(:) = totalUBound(:) - exclusiveUBound(:)
```

The *exclusive region* is determined during Array creation by the DistGrid argument. Optional arguments may be used to specify the *computational region* when the Array is created, by default it will be set equal to the exclusive region. The *total region*, i.e. the actual memory allocation for each DE, is also determined during Array creation. When creating the Array object from existing Fortran arrays the total region is set equal to the memory provided by the Fortran arrays. Otherwise the default is to allocate as much memory as is needed to accommodate the union of the DE-local exclusive and computational region. Finally it is also possible to use optional arguments to the ArrayCreate() call to specify the total region of the object explicitly.

The `ESMF_ArrayCreate()` call checks that the input parameters are consistent and will result in an Array that fulfills all of the above mentioned requirements for its DE-local regions.

Once an Array object has been created the exclusive and total regions are fixed. The computational region, however, may be adjusted within the limits of the total region using the `ArraySet()` call.

The *interior region* is very different from the other regions in that it cannot be specified. The *interior region* for each DE is a *consequence* of the choices made for the other regions collectively across all DEs into which an Array object is decomposed. An Array object can be queried for its DE-local *interior regions* as to offer additional information to the user necessary to write more efficient code.

By default the bounds of each DE-local *total region* are defined as to put the start of the DE-local *exclusive region* at the "origin" of the local index space, i.e. at `(1, 1, ..., 1)`. With that definition the following loop will access each element of the DE-local memory segment for each PET-local DE of the Array object used in the previous sections and print its content.

```
  do localDe=0, localDeCount-1
    call ESMF_LocalArrayGet(larrayList(localDe), myFarray, &
       datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)
    do i=1, size(myFarray, 1)
      do j=1, size(myFarray, 2)
        print *, "localPET=", localPet, " localDE=", &
            localDe, ": array(",i,",",j,")=", myFarray(i,j)
      enddo
    enddo
  enddo
```

28.2.7 Array bounds

The loop over Array elements at the end of the last section only works correctly because of the default definition of the *computational* and *total regions* used in the example. In general, without such specific knowledge about an Array object, it is necessary to use a more formal approach to access its regions with DE-local indices.

The DE-local *exclusive region* takes a central role in the definition of Array bounds. Even as the *computational region* may adjust during the course of execution the *exclusive region* remains unchanged. The *exclusive region* provides a unique reference frame for the index space of all Arrays associated with the same DistGrid.

There is a choice between two indexing options that needs to be made during Array creation. By default each DE-local exclusive region starts at `(1, 1, ..., 1)`. However, for some computational kernels it may be more convenient to choose the index bounds of the DE-local exclusive regions to match the index space coordinates as they are defined in the corresponding DistGrid object. The second option is only available if the DistGrid object does not contain any non-contiguous decompositions (such as cyclically decomposed dimensions).

The following example code demonstrates the safe way of dereferencing the DE-local exclusive regions of the previously created `array` object.

```
  allocate(exclusiveUBound(2, 0:localDeCount-1))  ! dimCount=2
  allocate(exclusiveLBound(2, 0:localDeCount-1))  ! dimCount=2
  call ESMF_ArrayGet(array, indexflag=indexflag, &
    exclusiveLBound=exclusiveLBound, exclusiveUBound=exclusiveUBound, rc=rc)
  if (indexflag == ESMF_INDEX_DELOCAL) then
    ! this is the default
!    print *, "DE-local exclusive regions start at (1,1)"
    do localDe=0, localDeCount-1
      call ESMF_LocalArrayGet(larrayList(localDe), myFarray, &
          datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)
      do i=1, exclusiveUBound(1, localDe)
        do j=1, exclusiveUBound(2, localDe)
!          print *, "DE-local exclusive region for localDE=", localDe, &
!            ": array(",i,",",j,")=", myFarray(i,j)
        enddo
      enddo
    enddo
  else if (indexflag == ESMF_INDEX_GLOBAL) then
    ! only if set during ESMF_ArrayCreate()
```

```
!    print *, "DE-local exclusive regions of this Array have global bounds"
    do localDe=0, localDeCount-1
      call ESMF_LocalArrayGet(larrayList(localDe), myFarray, &
         datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)
      do i=exclusiveLBound(1, localDe), exclusiveUBound(1, localDe)
        do j=exclusiveLBound(2, localDe), exclusiveUBound(2, localDe)
!          print *, "DE-local exclusive region for localDE=", localDe, &
!            ": array(",i,",",j,")=", myFarray(i,j)
        enddo
      enddo
    enddo
  endif
  call ESMF_ArrayDestroy(array, rc=rc) ! destroy the array object
```

Obviously the second branch of this simple code will work for either case, however, if a complex computational kernel was written assuming `ESMF_INDEX_DELOCAL` type bounds the second branch would simply be used to indicate the problem and bail out.

The advantage of the `ESMF_INDEX_GLOBAL` index option is that the Array bounds directly contain information on where the DE-local Array piece is located in a global index space sense. When the `ESMF_INDEX_DELOCAL` option is used the correspondence between local and global index space must be made by querying the associated DistGrid for the DE-local `indexList` arguments.

28.2.8 Computational region and extra elements for halo or padding

In the previous examples the computational region of `array` was chosen by default to be identical to the exclusive region defined by the DistGrid argument during Array creation. In the following the same `arrayspec` and `distgrid` objects as before will be used to create an Array but now a larger computational region shall be defined around each DE-local exclusive region. Furthermore, extra space will be defined around the computational region of each DE to accommodate a halo and/or serve as memory padding.

In this example the `indexflag` argument is set to `ESMF_INDEX_GLOBAL` indicating that the bounds of the exclusive region correspond to the index space coordinates as they are defined by the DistGrid object.

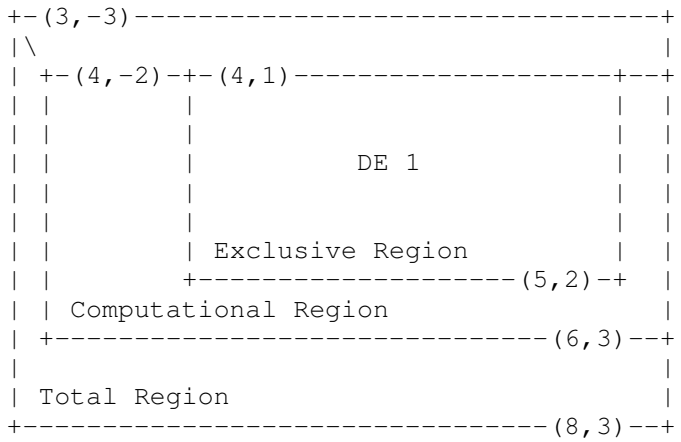
The same `arrayspec` and `distgrid` objects as before are used which also allows the reuse of the already allocated `larrayList` variable.

```
  array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
    computationalLWidth=(/0,3/), computationalUWidth=(/1,1/), &
    totalLWidth=(/1,4/), totalUWidth=(/3,1/), &
    indexflag=ESMF_INDEX_GLOBAL, rc=rc)
```

Obtain the `larrayList` on every PET.

```
  allocate(localDeToDeMap(0:localDeCount-1))
  call ESMF_ArrayGet(array, localarrayList=larrayList, &
    localDeToDeMap=localDeToDeMap, rc=rc)
```

The bounds of DE 1 for `array` are shown in the following diagram to illustrate the situation. Notice that the `totalLWidth` and `totalUWidth` arguments in the ArrayCreate() call define the total region with respect to the exclusive region given for each DE by the `distgrid` argument.



When working with this `array` it is possible for the computational kernel to overstep the exclusive region for both read/write access (computational region) and potentially read-only access into the total region outside of the computational region, if a halo operation provides valid entries for these elements.

The Array object can be queried for absolute *bounds*

```
allocate(computationalLBound(2, 0:localDeCount-1))  ! dimCount=2
allocate(computationalUBound(2, 0:localDeCount-1))  ! dimCount=2
allocate(totalLBound(2, 0:localDeCount-1))          ! dimCount=2
allocate(totalUBound(2, 0:localDeCount-1))          ! dimCount=2
call ESMF_ArrayGet(array, exclusiveLBound=exclusiveLBound, &
  exclusiveUBound=exclusiveUBound, &
  computationalLBound=computationalLBound, &
  computationalUBound=computationalUBound, &
  totalLBound=totalLBound, &
  totalUBound=totalUBound, rc=rc)
```

or for the relative *widths*.

```
allocate(computationalLWidth(2, 0:localDeCount-1))  ! dimCount=2
allocate(computationalUWidth(2, 0:localDeCount-1))  ! dimCount=2
allocate(totalLWidth(2, 0:localDeCount-1))          ! dimCount=2
allocate(totalUWidth(2, 0:localDeCount-1))          ! dimCount=2
call ESMF_ArrayGet(array, computationalLWidth=computationalLWidth, &
  computationalUWidth=computationalUWidth, totalLWidth=totalLWidth, &
  totalUWidth=totalUWidth, rc=rc)
```

Either way the dereferencing of Array data is centered around the DE-local exclusive region:

```
do localDe=0, localDeCount-1
  call ESMF_LocalArrayGet(larrayList(localDe), myFarray, &
     datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)
  ! initialize the DE-local array
  myFarray = 0.1d0 * localDeToDeMap(localDe)
```

```
    ! first time through the total region of array
!    print *, "myFarray bounds for DE=", localDeToDeMap(localDe), &
!      lbound(myFarray),  ubound(myFarray)
    do j=exclusiveLBound(2, localDe), exclusiveUBound(2, localDe)
      do i=exclusiveLBound(1, localDe), exclusiveUBound(1, localDe)
!       print *, "Excl region DE=", localDeToDeMap(localDe), &
!       ": array(",i,",",j,")=",  myFarray(i,j)
      enddo
    enddo
    do j=computationalLBound(2, localDe), computationalUBound(2, localDe)
      do i=computationalLBound(1, localDe), computationalUBound(1, localDe)
!       print *, "Excl region DE=", localDeToDeMap(localDe), &
!       ": array(",i,",",j,")=", myFarray(i,j)
      enddo
    enddo
    do j=totalLBound(2, localDe), totalUBound(2, localDe)
      do i=totalLBound(1, localDe), totalUBound(1, localDe)
!       print *, "Total region DE=", localDeToDeMap(localDe), &
!       ": array(",i,",",j,")=", myFarray(i,j)
      enddo
    enddo

    ! second time through the total region of array
    do j=exclusiveLBound(2, localDe)-totalLWidth(2, localDe), &
      exclusiveUBound(2, localDe)+totalUWidth(2, localDe)
      do i=exclusiveLBound(1, localDe)-totalLWidth(1, localDe), &
        exclusiveUBound(1, localDe)+totalUWidth(1, localDe)
!       print *, "Excl region DE=", localDeToDeMap(localDe), &
!       ": array(",i,",",j,")=", myFarray(i,j)
      enddo
    enddo
  enddo
```

28.2.9 Create 1D and 3D Arrays

All previous examples were written for the 2D case. There is, however, no restriction within the Array or DistGrid class that limits the dimensionality of Array objects beyond the language-specific limitations (7D for Fortran).

In order to create an `n`-dimensional Array the rank indicated by both the `arrayspec` and the `distgrid` arguments specified during Array create must be equal to `n`. A 1D Array of double precision real data hence requires the following `arrayspec`.

```
  call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=1, rc=rc)
```

The index space covered by the Array and the decomposition description is provided to the Array create method by the `distgrid` argument. The index space in this example has 16 elements and covers the interval $[-10, 5]$. It is decomposed into as many DEs as there are PETs in the current context.

```
  distgrid1D = ESMF_DistGridCreate(minIndex=(/-10/), maxIndex=(/5/), &
    regDecomp=(/petCount/), rc=rc)
```

A 1D Array object with default regions can now be created.

```
array1D = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid1D, rc=rc)
```

The creation of a 3D Array proceeds analogous to the 1D case. The rank of the `arrayspec` must be changed to 3

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=3, rc=rc)
```

and an appropriate 3D DistGrid object must be created

```
distgrid3D = ESMF_DistGridCreate(minIndex=(/1,1,1/), &
  maxIndex=(/16,16,16/), regDecomp=(/4,4,4/), rc=rc)
```

before an Array object can be created.

```
array3D = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid3D, rc=rc)
```

The `distgrid3D` object decomposes the 3-dimensional index space into $4 \times 4 \times 4 = 64$ DEs. These DEs are laid out across the computational resources (PETs) of the current component according to a default DELayout that is created during the DistGrid create call. Notice that in the index space proposal a DELayout does not have a sense of dimensionality. The DELayout function is simply to map DEs to PETs. The DistGrid maps chunks of index space against DEs and thus its rank is equal to the number of index space dimensions.

The previously defined DistGrid and the derived Array object decompose the index space along all three dimension. It is, however, not a requirement that the decomposition be along all dimensions. An Array with the same 3D index space could as well be decomposed along just one or along two of the dimensions. The following example shows how for the same index space only the last two dimensions are decomposed while the first Array dimension has full extent on all DEs.

```
call ESMF_ArrayDestroy(array3D, rc=rc)
call ESMF_DistGridDestroy(distgrid3D, rc=rc)
distgrid3D = ESMF_DistGridCreate(minIndex=(/1,1,1/), &
  maxIndex=(/16,16,16/), regDecomp=(/1,4,4/), rc=rc)
array3D = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid3D, rc=rc)
```

28.2.10 Working with Arrays of different rank

Assume a computational kernel that involves the `array3D` object as it was created at the end of the previous section. Assume further that the kernel also involves a 2D Array on a 16x16 index space where each point (j,k) was interacting with each (i,j,k) column of the 3D Array. An efficient formulation would require that the decomposition of the 2D Array must match that of the 3D Array and further the DELayout be identical. The following code shows how this can be accomplished.

```
call ESMF_DistGridGet(distgrid3D, delayout=delayout, rc=rc) ! get DELayout
distgrid2D = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/16,16/), &
```

```
    regDecomp=(/4,4/), delayout=delayout, rc=rc)
  call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)
  array2D = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid2D, rc=rc)
```

Now the following kernel is sure to work with `array3D` and `array2D`.

```
  call ESMF_DELayoutGet(delayout, localDeCount=localDeCount, rc=rc)
  allocate(larrayList1(0:localDeCount-1))
  call ESMF_ArrayGet(array3D, localarrayList=larrayList1, rc=rc)
  allocate(larrayList2(0:localDeCount-1))
  call ESMF_ArrayGet(array2D, localarrayList=larrayList2, rc=rc)
  do localDe=0, localDeCount-1
    call ESMF_LocalArrayGet(larrayList1(localDe), myFarray3D, &
      datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)
    myFarray3D = 0.1d0 * localDe ! initialize
    call ESMF_LocalArrayGet(larrayList2(localDe), myFarray2D, &
      datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)
    myFarray2D = 0.5d0 * localDe ! initialize
    do k=1, 4
      do j=1, 4
        dummySum = 0.d0
        do i=1, 16
          dummySum = dummySum + myFarray3D(i,j,k) ! sum up the (j,k) column
        enddo
        dummySum = dummySum * myFarray2D(j,k) ! multiply with local 2D element
!        print *, "dummySum(",j,k,")=",dummySum
      enddo
    enddo
  enddo
```

28.2.11 Array and DistGrid rank – 2D+1 Arrays

Except for the special Array create interface that implements a copy from an existing Array object all other Array create interfaces require the specification of at least two arguments: `farray` and `distgrid`, `larrayList` and `distgrid`, or `arrayspec` and `distgrid`. In all these cases both required arguments contain a sense of dimensionality. The relationship between these two arguments deserves extra attention.

The first argument, `farray`, `larrayList` or `arrayspec`, determines the rank of the created Array object, i.e. the dimensionality of the actual data storage. The rank of a native language array, extracted from an Array object, is equal to the rank specified by either of these arguments. So is the `rank` that is returned by the `ESMF_ArrayGet()` call.

The rank specification contained in the `distgrid` argument, which is of type `ESMF_DistGrid`, on the other hand has no effect on the rank of the Array. The `dimCount` specified by the DistGrid object, which may be equal, greater or less than the Array rank, determines the dimensionality of the *decomposition*.

While there is no constraint between DistGrid `dimCount` and Array `rank`, there is an important relationship between the two, resulting in the concept of index space dimensionality. Array dimensions can be arbitrarily mapped against DistGrid dimension, rendering them *decomposed* dimensions. The index space dimensionality is equal to the number of decomposed Array dimensions.

Array dimensions that are not mapped to DistGrid dimensions are the *undistributed* dimensions of the Array. They are not part of the index space. The mapping is specified during `ESMF_ArrayCreate()` via the `distgridToArrayMap` argument. DistGrid dimensions that have not been associated with Array dimensions are *replicating* dimensions. The Array will be replicated across the DEs that lie along replication DistGrid dimensions.

Undistributed Array dimensions can be used to store multi-dimensional data for each Array index space element. One application of this is to store the components of a vector quantity in a single Array. The same 2D `distgrid` object as before will be used.

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
  regDecomp=(/2,3/), rc=rc)
```

The rank in the `arrayspec` argument, however, must change from 2 to 3 in order to provide for the extra Array dimension.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=3, rc=rc)
```

During Array creation with extra dimension(s) it is necessary to specify the bounds of these undistributed dimension(s). This requires two additional arguments, `undistLBound` and `undistUBound`, which are vectors in order to accommodate multiple undistributed dimensions. The other arguments remain unchanged and apply across all undistributed components.

```
array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
  totalLWidth=(/0,1/), totalUWidth=(/0,1/), &
  undistLBound=(/1/), undistUBound=(/2/), rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

This will create `array` with 2+1 dimensions. The 2D DistGrid is used to describe decomposition into DEs with 2 Array dimensions mapped to the DistGrid dimensions resulting in a 2D index space. The extra Array dimension provides storage for multi component user data within the Array object.

By default the `distgrid` dimensions are associated with the first Array dimensions in sequence. For the example above this means that the first 2 Array dimensions are decomposed according to the provided 2D DistGrid. The 3rd Array dimension does not have an associated DistGrid dimension, rendering it an undistributed Array dimension.

Native language access to an Array with undistributed dimensions is in principle the same as without extra dimensions.

```
call ESMF_ArrayGet(array, localDeCount=localDeCount, rc=rc)
allocate(larrayList(0:localDeCount-1))
call ESMF_ArrayGet(array, localarrayList=larrayList, rc=rc)
```

The following loop shows how a Fortran pointer to the DE-local data chunks can be obtained and used to set data values in the exclusive regions. The `myFarray3D` variable must be of rank 3 to match the Array rank of `array`. However, variables such as `exclusiveUBound` that store the information about the decomposition, remain to be allocated for the 2D index space.

```
call ESMF_ArrayGet(array, exclusiveLBound=exclusiveLBound, &
  exclusiveUBound=exclusiveUBound, rc=rc)
do localDe=0, localDeCount-1
  call ESMF_LocalArrayGet(larrayList(localDe), myFarray3D, &
     datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)
  myFarray3D = 0.0 ! initialize
  myFarray3D(exclusiveLBound(1,localDe):exclusiveUBound(1,localDe), &
```

```
    exclusiveLBound(2,localDe):exclusiveUBound(2,localDe), &
    1) = 5.1 ! dummy assignment
  myFarray3D(exclusiveLBound(1,localDe):exclusiveUBound(1,localDe), &
    exclusiveLBound(2,localDe):exclusiveUBound(2,localDe), &
    2) = 2.5 ! dummy assignment
enddo
deallocate(larrayList)
```

For some applications the default association rules between DistGrid and Array dimensions may not satisfy the user's needs. The optional `distgridToArrayMap` argument can be used during Array creation to explicitly specify the mapping between DistGrid and Array dimensions. To demonstrate this the following lines of code reproduce the above example but with rearranged dimensions. Here the `distgridToArrayMap` argument is a list with two elements corresponding to the DistGrid `dimCount` of 2. The first element indicates which Array dimension the first DistGrid dimension is mapped against. Here the 1st DistGrid dimension maps against the 3rd Array dimension and the 2nd DistGrid dimension maps against the 1st Array dimension. This leaves the 2nd Array dimension to be the extra and undistributed dimension in the resulting Array object.

```
call ESMF_ArrayDestroy(array, rc=rc)
array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
  distgridToArrayMap=(/3, 1/), totalLWidth=(/0,1/), totalUWidth=(/0,1/), &
  undistLBound=(/1/), undistUBound=(/2/), rc=rc)
```

Operations on the Array object as a whole are unchanged by the different mapping of dimensions.

When working with Arrays that contain explicitly mapped Array and DistGrid dimensions it is critical to know the order in which the entries of *width* and *bound* arguments that are associated with distributed Array dimensions are specified. The size of these arguments is equal to the DistGrid `dimCount`, because the maximum number of distributed Array dimensions is given by the dimensionality of the index space.

The order of dimensions in these arguments, however, is *not* that of the associated DistGrid. Instead each entry corresponds to the distributed Array dimensions in sequence. In the example above the entries in `totalLWidth` and `totalUWidth` correspond to Array dimensions 1 and 3 in this sequence.

The `distgridToArrrayMap` argument optionally provided during Array create indicates how the DistGrid dimensions map to Array dimensions. The inverse mapping, i.e. Array to DistGrid dimensions, is just as important. The `ESMF_ArrayGet()` call offers both mappings as `distgridToArrrayMap` and `arrayToDistGridMap`, respectively. The number of elements in `arrayToDistGridMap` is equal to the rank of the Array. Each element corresponds to an Array dimension and indicates the associated DistGrid dimension by an integer number. An entry of "0" in `arrayToDistGridMap` indicates that the corresponding Array dimension is undistributed.

Correct understanding about the association between Array and DistGrid dimensions becomes critical for correct data access into the Array.

```
allocate(arrayToDistGridMap(3))  ! arrayRank = 3
call ESMF_ArrayGet(array, arrayToDistGridMap=arrayToDistGridMap, &
  exclusiveLBound=exclusiveLBound, exclusiveUBound=exclusiveUBound, &
  localDeCount=localDeCount, rc=rc)
if (arrayToDistGridMap(2) /= 0) then   ! check if extra dimension at
  ! expected index indicate problem and bail out
endif
! obtain larrayList for local DEs
allocate(larrayList(0:localDeCount-1))
call ESMF_ArrayGet(array, localarrayList=larrayList, rc=rc)
do localDe=0, localDeCount-1
```

```
  call ESMF_LocalArrayGet(larrayList(localDe), myFarray3D, &
     datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)
  myFarray3D(exclusiveLBound(1,localDe):exclusiveUBound(1,localDe), &
    1, exclusiveLBound(2,localDe):exclusiveUBound(2, &
    localDe)) = 10.5 !dummy assignment
  myFarray3D(exclusiveLBound(1,localDe):exclusiveUBound(1,localDe), &
    2, exclusiveLBound(2,localDe):exclusiveUBound(2, &
    localDe)) = 23.3 !dummy assignment
enddo
deallocate(exclusiveLBound, exclusiveUBound)
deallocate(arrayToDistGridMap)
deallocate(larrayList)
call ESMF_ArrayDestroy(array, rc=rc)
if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
```

28.2.12 Arrays with replicated dimensions

Thus far most examples demonstrated cases where the DistGrid `dimCount` was equal to the Array `rank`. The previous section introduced the concept of Array *tensor* dimensions when `dimCount < rank`. In this section `dimCount` and `rank` are assumed completely unconstrained and the relationship to `distgridToArrayMap` and `arrayToDistGridMap` will be discussed.

The Array class allows completely arbitrary mapping between Array and DistGrid dimensions. Most cases considered in the previous sections used the default mapping which assigns the DistGrid dimensions in sequence to the lower Array dimensions. Extra Array dimensions, if present, are considered non-distributed tensor dimensions for which the optional `undistLBound` and `undistUBound` arguments must be specified.

The optional `distgridToArrayMap` argument provides the option to override the default DistGrid to Array dimension mapping. The entries of the `distgridToArrayMap` array correspond to the DistGrid dimensions in sequence and assign a unique Array dimension to each DistGrid dimension. DistGrid and Array dimensions are indexed starting at `1` for the lowest dimension. A value of `"0"` in the `distgridToArrayMap` array indicates that the respective DistGrid dimension is *not* mapped against any Array dimension. What this means is that the Array will be replicated along this DistGrid dimension.

As a first example consider the case where a 1D Array

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=1, rc=rc)
```

is created on the 2D DistGrid used during the previous section.

```
array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, rc=rc)
```

Here the default DistGrid to Array dimension mapping is used which assigns the Array dimensions in sequence to the DistGrid dimensions starting with dimension "1". Extra DistGrid dimensions are considered replicator dimensions because the Array will be replicated along those dimensions. In the above example the 2nd DistGrid dimension will cause 1D Array pieces to be replicated along the DEs of the 2nd DistGrid dimension. Replication in the context of `ESMF_ArrayCreate()` does not mean that data values are communicated and replicated between different DEs, but it means that different DEs provide memory allocations for *identical* exclusive elements.

Access to the data storage of an Array that has been replicated along DistGrid dimensions is the same as for Arrays without replication.

```
call ESMF_ArrayGet(array, localDeCount=localDeCount, rc=rc)


allocate(larrayList(0:localDeCount-1))
allocate(localDeToDeMap(0:localDeCount-1))
call ESMF_ArrayGet(array, localarrayList=larrayList, &
  localDeToDeMap=localDeToDeMap, rc=rc)
```

The `array` object was created without additional padding which means that the bounds of the Fortran array pointer correspond to the bounds of the exclusive region. The following loop will cycle through all local DEs, print the DE number as well as the Fortran array pointer bounds. The bounds should be:

```
           lbound       ubound

 DE 0:       1            3         --+
 DE 2:       1            3         --|  1st replication set
 DE 4:       1            3         --+

 DE 1:       1            2         --+
 DE 3:       1            2         --|  2nd replication set
 DE 5:       1            2         --+
```

```
do localDe=0, localDeCount-1
  call ESMF_LocalArrayGet(larrayList(localDe), myFarray1D, &
    datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)


  print *, "localPet: ", localPet, "DE ",localDeToDeMap(localDe)," [", &
    lbound(myFarray1D), ubound(myFarray1D),"]"
enddo
deallocate(larrayList)
deallocate(localDeToDeMap)
call ESMF_ArrayDestroy(array, rc=rc)
```

The Fortran array pointer in the above loop was of rank 1 because the Array object was of rank 1. However, the `distgrid` object associated with `array` is 2-dimensional! Consequently DistGrid based information queried from `array` will be 2D. The `distgridToArrayMap` and `arrayToDistGridMap` arrays provide the necessary mapping to correctly associate DistGrid based information with Array dimensions.

The next example creates a 2D Array

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)
```

on the previously used 2D DistGrid. By default, i.e. without the `distgridToArrayMap` argument, both DistGrid dimensions would be associated with the two Array dimensions. However, the `distgridToArrayMap` specified in the following call will only associate the second DistGrid dimension with the first Array dimension. This will render the first DistGrid dimension a replicator dimension and the second Array dimension a tensor dimension for which 1D `undistLBound` and `undistUBound` arguments must be supplied.

```
array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
  distgridToArrayMap=(/0,1/), undistLBound=(/11/), &
  undistUBound=(/14/), rc=rc)


call ESMF_ArrayDestroy(array, rc=rc)
```

Finally, the same `arrayspec` and `distgrid` arguments are used to create a 2D Array that is fully replicated in both dimensions of the DistGrid. Both Array dimensions are now tensor dimensions and both DistGrid dimensions are replicator dimensions.

```
array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
  distgridToArrayMap=(/0,0/), undistLBound=(/11,21/), &
  undistUBound=(/14,22/), rc=rc)
```

The result will be an Array with local lower bound (/11,21/) and upper bound (/14,22/) on all 6 DEs of the DistGrid.

```
call ESMF_ArrayDestroy(array, rc=rc)


call ESMF_DistGridDestroy(distgrid, rc=rc)
```

Replicated Arrays can also be created from existing local Fortran arrays. The following Fortran array allocation will provide a 3 x 10 array on each PET.

```
allocate(myFarray2D(3,10))
```

Assuming a petCount of 4 the following DistGrid defines a 2D index space that is distributed across the PETs along the first dimension.

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/40,10/), rc=rc)
```

The following call creates an Array object on the above distgrid using the locally existing `myFarray2D` Fortran arrays. The difference compared to the case with automatic memory allocation is that instead of `arrayspec` the Fortran array is provided as argument. Furthermore, the `undistLBound` and `undistUBound` arguments can be omitted, defaulting into Array tensor dimension lower bound of 1 and an upper bound equal to the size of the respective Fortran array dimension.

```
array = ESMF_ArrayCreate(farray=myFarray2D, distgrid=distgrid, &
  indexflag=ESMF_INDEX_DELOCAL, distgridToArrayMap=(/0,2/), rc=rc)
```

The `array` object associates the 2nd DistGrid dimension with the 2nd Array dimension. The first DistGrid dimension is not associated with any Array dimension and will lead to replication of the Array along the DEs of this direction.

```
call ESMF_ArrayDestroy(array, rc=rc)


call ESMF_DistGridDestroy(distgrid, rc=rc)
```

28.2.13 Shared memory features: DE pinning, sharing, and migration

Practically all modern computer systems today utilize multi-core processors, supporting the concurrent execution of multiple hardware threads. A number of these multi-core processors are commonly packaged into the same compute node, having access to the same physical memory. Under ESMF each hardware thread (or core) is identified as a unique Processing Element (PE). The collection of PEs that share the same physical memory (i.e. compute node) is referred to as a Single System Image (SSI). The ESMF Array class implements features that allow the user to leverage the shared memory within each SSI to efficiently exchange data without copies or explicit communication calls.

The software threads executing an ESMF application on the hardware, and that ESMF is aware of, are referred to as Persistent Execution Threads (PETs). In practice a PET can typically be thought of as an MPI rank, i.e. an OS process, defining its own private virtual address space (VAS). The ESMF Virtual Machine (VM) class keeps track of the mapping between PETs and PEs, and their location on the available SSIs.

When an ESMF Array object is created, the specified DistGrid indicates how many Decomposition Elements (DEs) are created. Each DE has its own memory allocation to hold user data. The DELayout, referenced by the DistGrid, determines which PET is considered the *owner* of each of the DEs. Queried for the local DEs, the Array object returns the list of DEs that are owned by the local PET making the query.

By default DEs are *pinned* to the PETs under which they were created. The memory allocation associated with a specific DE is only defined in the VAS of the PET to which the DE is pinned. As a consequence, only the PET owning a DE has access to its memory allocation.

On shared memory systems, however, ESMF allows DEs to be pinned to SSIs instead of PETs. In this case the PET under which a DE was created is still consider the owner, but now *all* PETs under the same SSI have access to the DE. For this the memory allocation associated with the DE is mapped into the VAS of all the PETs under the SSI.

To create an Array with each DE pinned to SSI instead of PET, first query the VM for the available level of support.

```
call ESMF_VMGet(vm, ssiSharedMemoryEnabledFlag=ssiSharedMemoryEnabled, rc=rc)


if (ssiSharedMemoryEnabled) then
```

Knowing that the SSI shared memory feature is available, it is now possible to create an Array object with DE to SSI pinning.

```
  distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/40,10/), rc=rc)


  array = ESMF_ArrayCreate(typekind=ESMF_TYPEKIND_R8, distgrid=distgrid, &
    pinflag=ESMF_PIN_DE_TO_SSI, rc=rc)
```

Just as in the cases discussed before, where the same DistGrid was used, a default DELayout with as many DEs as PETs in the VM is constructed. Setting the `pinflag` to `ESMF_PIN_DE_TO_SSI` does not change the fact that each

PET owns exactly one of the DEs. However, assuming that this code is run on a set of PETs that are all located under the same SSI, every PET now has *access* to all of the DEs. The situation can be observed by querying for both the `localDeCount`, and the `ssiLocalDeCount`.

```
call ESMF_ArrayGet(array, localDeCount=localDeCount, &
  ssiLocalDeCount=ssiLocalDeCount, rc=rc)
```

Assuming execution on 4 PETs, all located on the same SSI, the values of the returned variable are `localDeCount==1` and `ssiLocalDeCount==4` on all of the PETs. The mapping between each PET's local DE, and the global DE index is provided through the `localDeToDeMap` array argument. The amount of mapping information returned is dependent on how large `localDeToDeMap` has been sized by the user. For `size(localDeToDeMap)==localDeCount`, only mapping information for those DEs *owned* by the local PET is filled in. However for `size(localDeToDeMap)==ssiLocalDeCount`, mapping information for all locally *accessible* DEs is returned, including those owned by other PETs on the same SSI.

```
allocate(localDeToDeMap(0:ssiLocalDeCount-1))
call ESMF_ArrayGet(array, localDeToDeMap=localDeToDeMap, rc=rc)
```

The first `localDeCount` entries of `localDeToDeMap` are always the global DE indices of the DEs *owned* by the local PET. The remaining `ssiLocalDeCount-localDeCount` entries are the global DE indices of DEs *shared* by other PETs. The ordering of the shared DEs is from smallest to greatest, excluding the locally owned DEs, which were already listed at the beginning of `localDeToDeMap`. For the current case, again assuming execution on 4 PETs all located on the same SSI, we expect the following situation:

PET 0: `localDeToDeMap==(/0,1,2,3/)`
PET 1: `localDeToDeMap==(/1,0,2,3/)`
PET 2: `localDeToDeMap==(/2,0,1,3/)`
PET 3: `localDeToDeMap==(/3,0,1,2/)`

Each PET can access the memory allocations associated with *all* of the DEs listed in the `localDeToDeMap` returned by the Array object. Direct access to the Fortran array pointer of a specific memory allocation is available through `ESMF_ArrayGet()`. Here each PET queries for the `farrayPtr` of `localDe==2`, i.e. the 2nd shared DE.

```
call ESMF_ArrayGet(array, farrayPtr=myFarray, localDe=2, rc=rc)
```

Now variable `myFarray` on PETs 0 and 1 both point to the *same* memory allocation for global DE 2. Both PETs have access to the same piece of shared memory! The same is true for PETs 2 and 3, pointing to the shared memory allocation of global DE 1.

It is important to note that all of the typical considerations surrounding shared memory programming apply when accessing shared DEs! Proper synchronization between PETs accessing shared DEs is critical to avoid *race conditions*. Also performance issues like *false sharing* need to be considered for optimal use.

For a simple demonstration, PETs 0 and 2 fill the entire memory allocation of DE 2 and 1, respectively, to a unique value.

```
if (localPet==0) then
  myFarray = 12345.6789d0
else if (localPet==2) then
  myFarray = 6789.12345d0
endif
```

Here synchronization is needed before any PETs that share access to the same DEs can safely access the data without race condition. The Array class provides a simple synchronization method that can be used.

```
call ESMF_ArraySync(array, rc=rc) ! prevent race condition
```

Now it is safe for PETs 1 and 3 to access the shared DEs. We expect to find the data that was set above. For simplicity of the code only the first array element is inspected here.

```
if (localPet==1) then
  if (abs(myFarray(1,1)-12345.6789d0)>1.d10) print *, "bad data detected"
else if (localPet==3) then
  if (abs(myFarray(1,1)-6789.12345d0)>1.d10) print *, "bad data detected"
endif
```

Working with shared DEs requires additional bookkeeping on the user code level. In some situations, however, DE sharing is simply used as a mechanism to *move* DEs between PETs without requiring data copies. One practical application of this case is the transfer of an Array between two components, both of which use the same PEs, but run with different number of PETs. These would typically be sequential components that use OpenMP on the user level with varying threading levels.

DEs that are pinned to SSI can be moved or *migrated* to any PET within the SSI. This is accomplished by creating a new Array object from an existing Array that was created with `pinflag=ESMF_PIN_DE_TO_SSI`. The information of how the DEs are to migrate between the old and the new Array is provided through a DELayout object. This object must have the same number of DEs and describes how they map to the PETs on the current VM. If this is in the context of a different component, the number of PETs might differ from the original VM under which the existing Array was created. This situation is explicitly supported, still the number of DEs must match.

Here a simple DELayout is created on the same 4 PETs, but with rotated DE ownerships:

DE 0 -> PET 1 (old PET 0)
DE 1 -> PET 2 (old PET 1)
DE 2 -> PET 3 (old PET 2)
DE 3 -> PET 0 (old PET 3)

```
delayout = ESMF_DELayoutCreate(petMap=(/1,2,3,0/), rc=rc) ! DE->PET mapping
```

The creation of the new Array is done by reference, i.e. `datacopyflag=ESMF_DATACOPY_REFERENCE`, since the new Array does not create its own memory allocations. Instead the new Array references the shared memory resources held by the incoming Array object.

```
arrayMigrated = ESMF_ArrayCreate(array, delayout=delayout, &
  datacopyflag=ESMF_DATACOPY_REFERENCE, rc=rc)
```

Querying `arrayMigrated` for the number of local DEs will return 1 on each PET. Sizing the `localDeToDeMap` accordingly and querying for it.

```
deallocate(localDeToDeMap) ! free previous allocation
allocate(localDeToDeMap(0:1))
call ESMF_ArrayGet(arrayMigrated, localDeToDeMap=localDeToDeMap, rc=rc)
```

This yields the following expected outcome:

PET 0: `localDeToDeMap==(/1/)`
PET 1: `localDeToDeMap==(/2/)`
PET 2: `localDeToDeMap==(/3/)`
PET 3: `localDeToDeMap==(/0/)`

On each PET the respective Fortran array pointer is returned by the Array.

```
    call ESMF_ArrayGet(arrayMigrated, farrayPtr=myFarray, rc=rc)
```

The same situation could have been achieved with the original `array`. However, it would have required first finding the correct local DE for the target global DE on each PET, and then querying `array` accordingly. If needed more repeatedly, this bookkeeping would need to be kept in a user code data structure. The DE migration feature on the other hand provides a formal way to create a standard ESMF Array object that can be used directly in any Array level method as usual, letting ESMF handle the extra bookkeeping needed.

Before destroying an Array whose DEs are shared between PETs, it is advisable to issue one more synchronization. This prevents cases where a PET still might be accessing a shared DE, while the owner PET is already destroying the Array, therefore deallocating the shared memory resource.

```
    call ESMF_ArraySync(array, rc=rc) ! prevent race condition


    call ESMF_ArrayDestroy(array, rc=rc)
```

Remember that `arrayMigrated` shares the same memory allocations that were held by `array`. Array `arrayMigrated` must therefore not be used beyond the life time of `array`. Best to destroy it now.

```
    call ESMF_ArrayDestroy(arrayMigrated, rc=rc)


  endif ! ending the ssiSharedMemoryEnabled conditional
```

28.2.14 Communication – Scatter and Gather

It is a common situation, particularly in legacy code, that an ESMF Array object must be filled with data originating from a large Fortran array stored on a single PET.

```
  if (localPet == 0) then
    allocate(farray(10,20,30))
    do k=1, 30
      do j=1, 20
        do i=1, 10
          farray(i, j, k) = k*1000 + j*100 +  i
        enddo
      enddo
```

```
  enddo
else
  allocate(farray(0,0,0))
endif


distgrid = ESMF_DistGridCreate(minIndex=(/1,1,1/), maxIndex=(/10,20,30/), &
  rc=rc)


call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_I4, rank=3, rc=rc)


array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, rc=rc)
```

The `ESMF_ArrayScatter()` method provides a convenient way of scattering array data from a single root PET across the DEs of an ESMF Array object.

```
call ESMF_ArrayScatter(array, farray=farray, rootPet=0, rc=rc)


deallocate(farray)
```

The destination of the ArrayScatter() operation are all the DEs of a single tile. For multi-tile Arrays the destination tile can be specified. The shape of the scattered Fortran array must match the shape of the destination tile in the ESMF Array.

Gathering data decomposed and distributed across the DEs of an ESMF Array object into a single Fortran array on root PET is accomplished by calling `ESMF_ArrayGather()`.

```
if (localPet == 3) then
  allocate(farray(10,20,30))
else
  allocate(farray(0,0,0))
endif

call ESMF_ArrayGather(array, farray=farray, rootPet=3, rc=rc)


deallocate(farray)
```

The source of the ArrayGather() operation are all the DEs of a single tile. For multi-tile Arrays the source tile can be specified. The shape of the gathered Fortran array must match the shape of the source tile in the ESMF Array.

The `ESMF_ArrayScatter()` operation allows to fill entire replicated Array objects with data coming from a single root PET.

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/5,5/), &
  regDecomp=(/2,3/), rc=rc)
```

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)


array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
  distgridToArrayMap=(/0,0/), undistLBound=(/11,21/), &
  undistUBound=(/14,22/), rc=rc)
```

The shape of the Fortran source array used in the Scatter() call must be that of the contracted Array, i.e. contracted DistGrid dimensions do not count. For the `array` just created this means that the source array on `rootPet` must be of shape 4 x 2.

```
if (localPet == 0) then
  allocate(myFarray2D(4,2))
  do j=1,2
    do i=1,4
      myFarray2D(i,j) = i * 100.d0 + j * 1.2345d0 ! initialize
    enddo
  enddo
else
  allocate(myFarray2D(0,0))
endif

call ESMF_ArrayScatter(array, farray=myFarray2D, rootPet=0, rc=rc)


deallocate(myFarray2D)
```

This will have filled each local 4 x 2 Array piece with the replicated data of `myFarray2D`.

```
call ESMF_ArrayDestroy(array, rc=rc)


call ESMF_DistGridDestroy(distgrid, rc=rc)
```

As a second example for the use of Scatter() and Gather() consider the following replicated Array created from existing local Fortran arrays.

```
allocate(myFarray2D(3,10))
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/40,10/), rc=rc)


array = ESMF_ArrayCreate(farray=myFarray2D, distgrid=distgrid, &
  indexflag=ESMF_INDEX_DELOCAL, distgridToArrayMap=(/0,2/), rc=rc)
```

The `array` object associates the 2nd DistGrid dimension with the 2nd Array dimension. The first DistGrid dimension is not associated with any Array dimension and will lead to replication of the Array along the DEs of this direction. Still, the local arrays that comprise the `array` object refer to independent pieces of memory and can be initialized independently.

```
myFarray2D = localPet ! initialize
```

However, the notion of replication becomes visible when an array of shape 3 x 10 on root PET 0 is scattered across the Array object.

```
if (localPet == 0) then
  allocate(myFarray2D2(5:7,11:20))

  do j=11,20
    do i=5,7
      myFarray2D2(i,j) = i * 100.d0 + j * 1.2345d0 ! initialize
    enddo
  enddo
else
  allocate(myFarray2D2(0,0))
endif

call ESMF_ArrayScatter(array, farray=myFarray2D2, rootPet=0, rc=rc)


deallocate(myFarray2D2)
```

The Array pieces on every DE will receive the same source data, resulting in a replication of data along DistGrid dimension 1.

When the inverse operation, i.e. `ESMF_ArrayGather()`, is applied to a replicated Array an intrinsic ambiguity needs to be considered. ESMF defines the gathering of data of a replicated Array as the collection of data originating from the numerically higher DEs. This means that data in replicated elements associated with numerically lower DEs will be ignored during `ESMF_ArrayGather()`. For the current example this means that changing the Array contents on PET 1, which here corresponds to DE 1,

```
if (localPet == 1) then
  myFarray2D = real(1.2345, ESMF_KIND_R8)
endif
```

will *not* affect the result of

```
allocate(myFarray2D2(3,10))
myFarray2D2 = 0.d0    ! initialize to a known value
call ESMF_ArrayGather(array, farray=myFarray2D2, rootPet=0, rc=rc)
```

The result remains completely defined by the unmodified values of Array in DE 3, the numerically highest DE. However, overriding the DE-local Array piece on DE 3

```
if (localPet==3) then
  myFarray2D = real(5.4321, ESMF_KIND_R8)
endif
```

will change the outcome of

```
call ESMF_ArrayGather(array, farray=myFarray2D2, rootPet=0, rc=rc)
```

as expected.

```
deallocate(myFarray2D2)

call ESMF_ArrayDestroy(array, rc=rc)


call ESMF_DistGridDestroy(distgrid, rc=rc)
```

28.2.15 Communication – Halo

One of the most fundamental communication pattern in domain decomposition codes is the *halo* operation. The ESMF Array class supports halos by allowing memory for extra elements to be allocated on each DE. See sections 28.2.2 and 28.2.8 for examples and details on how to create an Array with extra DE-local elements.

Here we consider an Array object that is created on a DistGrid that defines a 10 x 20 index space, decomposed into 4 DEs using a regular 2 x 2 decomposition.

```
distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/10,20/), &
  regDecomp=(/2,2/), rc=rc)
```

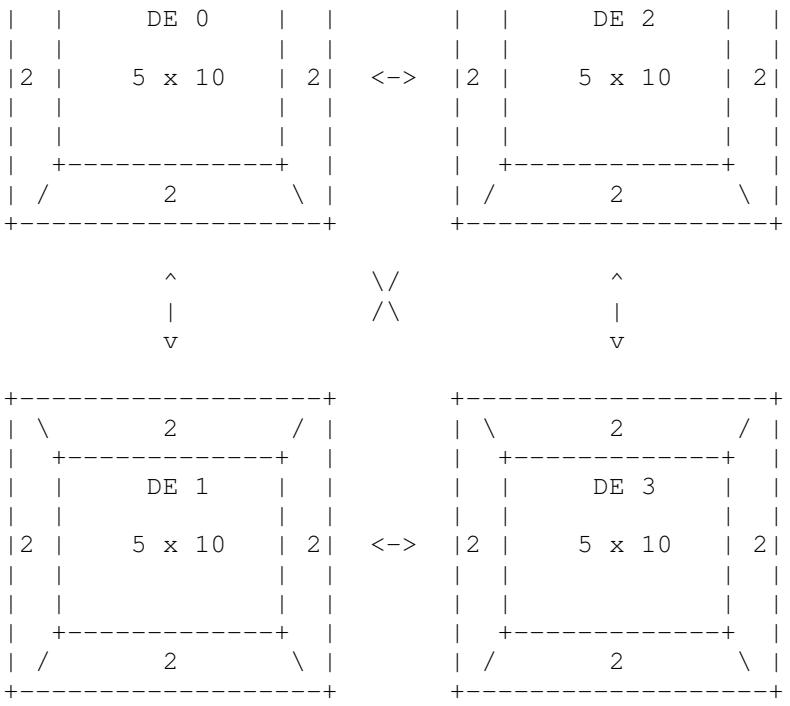
The Array holds 2D double precision float data.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)
```

The `totalLWidth` and `totalUWidth` arguments are used during Array creation to allocate 2 extra elements along every direction outside the exclusive region defined by the DistGrid for every DE. (The `indexflag` set to `ESMF_INDEX_GLOBAL` in this example does not affect the halo behavior of Array. The setting is simply more convenient for the following code.)

```
array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
  totalLWidth=(/2,2/), totalUWidth=(/2,2/), indexflag=ESMF_INDEX_GLOBAL, &
  rc=rc)
```

Without the explicit definition of boundary conditions in the DistGrid the following inner connections are defined.



The exclusive region on each DE is of shape 5 x 10, while the total region on each DE is of shape (5+2+2) x (10+2+2) = 9 x 14. In a typical application the elements in the exclusive region are updated exclusively by the PET that owns the DE. In this example the exclusive elements on every DE are initialized to the value $f(i,j)$ of the geometric function

$$f(i,j) = \sin(\alpha i)\cos(\beta j), \tag{1}$$

where

$$\alpha = 2\pi/N_i, i = 1, ...N_i \tag{2}$$

and

$$\beta = 2\pi/N_j, j = 1, ...N_j, \tag{3}$$

with $N_i = 10$ and $N_j = 20$.

```
a = 2. * 3.14159 / 10.
b = 2. * 3.14159 / 20.

call ESMF_ArrayGet(array, farrayPtr=farrayPtr, rc=rc)



call ESMF_ArrayGet(array, exclusiveLBound=eLB, exclusiveUBound=eUB, rc=rc)



do j=eLB(2,1), eUB(2,1)
  do i=eLB(1,1), eUB(1,1)
```

```
      farrayPtr(i,j) = sin(a*i) * cos(b*j)  ! test function
    enddo
  enddo
```

The above loop only initializes the exclusive elements on each DE. The extra elements, outside the exclusive region, are left untouched, holding undefined values. Elements outside the exclusive region that correspond to exclusive elements in neighboring DEs can be filled with the data values in those neighboring elements. This is the definition of the halo operation.

In ESMF the halo communication pattern is first precomputed and stored in a RouteHandle object. This RouteHandle can then be used repeatedly to perform the same halo operation in the most efficient way.

The default halo operation for an Array is precomputed by the following call.

```
call ESMF_ArrayHaloStore(array=array, routehandle=haloHandle, rc=rc)
```

The `haloHandle` now holds the default halo operation for `array`, which matches as many elements as possible outside the exclusive region to their corresponding halo source elements in neighboring DEs. Elements that could not be matched, e.g. at the edge of the global domain with open boundary conditions, will not be updated by the halo operation.

The `haloHandle` is applied through the `ESMF_ArrayHalo()` method.

```
call ESMF_ArrayHalo(array=array, routehandle=haloHandle, rc=rc)
```

Finally the resources held by `haloHandle` need to be released.

```
call ESMF_ArrayHaloRelease(routehandle=haloHandle, rc=rc)
```

The `array` object created above defines a 2 element wide rim around the exclusive region on each DE. Consequently the default halo operation used above will have resulted in updating both elements along the inside edges. For simple numerical kernels often a single halo element is sufficient. One way to achieve this would be to reduce the size of the rim surrounding the exclusive region to 1 element along each direction. However, if the same Array object is also used for higher order kernels during a different phase of the calculation, a larger element rim is required. For this case `ESMF_ArrayHaloStore()` offers two optional arguments `haloLDepth` and `haloUDepth`. Using these arguments a reduced halo depth can be specified.

```
call ESMF_ArrayHaloStore(array=array, routehandle=haloHandle, &
  haloLDepth=(/1,1/), haloUDepth=(/1,1/), rc=rc)
```

This halo operation with a depth of 1 is sufficient to support a simple quadratic differentiation kernel.

```
allocate(farrayTemp(eLB(1,1):eUB(1,1), eLB(2,1):eUB(2,1)))

do step=1, 4
  call ESMF_ArrayHalo(array=array, routehandle=haloHandle, rc=rc)
```

```
  do j=eLB(2,1), eUB(2,1)
    do i=eLB(1,1), eUB(1,1)
      if (i==1) then
        ! global edge
        farrayTemp(i,j) = 0.5 * (-farrayPtr(i+2,j) + 4.*farrayPtr(i+1,j) &
          - 3.*farrayPtr(i,j)) / a
      else if (i==10) then
        ! global edge
        farrayTemp(i,j) = 0.5 * (farrayPtr(i-2,j) - 4.*farrayPtr(i-1,j) &
          + 3.*farrayPtr(i,j)) / a
      else
        farrayTemp(i,j) = 0.5 * (farrayPtr(i+1,j) - farrayPtr(i-1,j)) / a
      endif
    enddo
  enddo
  farrayPtr(eLB(1,1):eUB(1,1), eLB(2,1):eUB(2,1)) = farrayTemp
enddo

deallocate(farrayTemp)

call ESMF_ArrayHaloRelease(routehandle=haloHandle, rc=rc)
```

The special treatment of the global edges in the above kernel is due to the fact that the underlying DistGrid object does not define any special boundary conditions. By default open global boundaries are assumed which means that the rim elements on the global edges are untouched during the halo operation, and cannot be used in the symmetric numerical derivative formula. The kernel can be simplified (and the calculation is more precise) with periodic boundary conditions along the first Array dimension.

First destroy the current Array and DistGrid objects.

```
call ESMF_ArrayDestroy(array, rc=rc)


call ESMF_DistGridDestroy(distgrid, rc=rc)
```

Create a DistGrid with periodic boundary condition along the first dimension.

```
allocate(connectionList(1))  ! one connection
call ESMF_DistGridConnectionSet(connection=connectionList(1), &
   tileIndexA=1, tileIndexB=1, positionVector=(/10, 0/), rc=rc)


distgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/10,20/), &
  regDecomp=(/2,2/), connectionList=connectionList, rc=rc)


deallocate(connectionList)
array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, &
  totalLWidth=(/2,2/), totalUWidth=(/2,2/), indexflag=ESMF_INDEX_GLOBAL, &
  rc=rc)
```

Initialize the exclusive elements to the same geometric function as before.

```
call ESMF_ArrayGet(array, farrayPtr=farrayPtr, rc=rc)



call ESMF_ArrayGet(array, exclusiveLBound=eLB, exclusiveUBound=eUB, rc=rc)



do j=eLB(2,1), eUB(2,1)
  do i=eLB(1,1), eUB(1,1)
    farrayPtr(i,j) = sin(a*i) * cos(b*j)  ! test function
  enddo
enddo
```

The numerical kernel only operates along the first dimension. An asymmetric halo depth can be used to take this fact into account.

```
call ESMF_ArrayHaloStore(array=array, routehandle=haloHandle, &
  haloLDepth=(/1,0/), haloUDepth=(/1,0/), rc=rc)
```

Now the same numerical kernel can be used without special treatment of global edge elements. The symmetric derivative formula can be used for all exclusive elements.

```
allocate(farrayTemp(eLB(1,1):eUB(1,1), eLB(2,1):eUB(2,1)))

do step=1, 4
  call ESMF_ArrayHalo(array=array, routehandle=haloHandle, rc=rc)


  do j=eLB(2,1), eUB(2,1)
    do i=eLB(1,1), eUB(1,1)
      farrayTemp(i,j) = 0.5 * (farrayPtr(i+1,j) - farrayPtr(i-1,j)) / a
    enddo
  enddo
  farrayPtr(eLB(1,1):eUB(1,1), eLB(2,1):eUB(2,1)) = farrayTemp
enddo
```

The precision of the above kernel can be improved by going to a higher order interpolation. Doing so requires that the halo depth must be increased. The following code resets the exclusive Array elements to the test function, precomputes a RouteHandle for a halo operation with depth 2 along the first dimension, and finally uses the deeper halo in the higher order kernel.

```
do j=eLB(2,1), eUB(2,1)
  do i=eLB(1,1), eUB(1,1)
```

```
    farrayPtr(i,j) = sin(a*i) * cos(b*j)  ! test function
  enddo
enddo

call ESMF_ArrayHaloStore(array=array, routehandle=haloHandle2, &
  haloLDepth=(/2,0/), haloUDepth=(/2,0/), rc=rc)



do step=1, 4
  call ESMF_ArrayHalo(array=array, routehandle=haloHandle2, rc=rc)


  do j=eLB(2,1), eUB(2,1)
    do i=eLB(1,1), eUB(1,1)
      farrayTemp(i,j) = (-farrayPtr(i+2,j) + 8.*farrayPtr(i+1,j) &
        - 8.*farrayPtr(i-1,j) + farrayPtr(i-2,j)) / (12.*a)
    enddo
  enddo
  farrayPtr(eLB(1,1):eUB(1,1), eLB(2,1):eUB(2,1)) = farrayTemp
enddo

deallocate(farrayTemp)
```

ESMF supports having multiple halo operations defined on the same Array object at the same time. Each operation can be accessed through its unique RouteHandle. The above kernel could have made `ESMF_ArrayHalo()` calls with a depth of 1 along the first dimension using the previously precomputed `haloHandle` if it needed to. Both RouteHandles need to release their resources when no longer used.

```
call ESMF_ArrayHaloRelease(routehandle=haloHandle, rc=rc)


call ESMF_ArrayHaloRelease(routehandle=haloHandle2, rc=rc)
```

Finally the Array and DistGrid objects can be destroyed.

```
call ESMF_ArrayDestroy(array, rc=rc)


call ESMF_DistGridDestroy(distgrid, rc=rc)
```

28.2.16 Communication – Halo for arbitrary distribution

In the previous section the Array *halo* operation was demonstrated for regularly decomposed ESMF Arrays. However, the ESMF halo operation is not restricted to regular decompositions. The same Array halo methods apply unchanged

to Arrays that are created on arbitrarily distributed DistGrids. This includes the non-blocking features discussed in section 28.2.20.

All of the examples in this section are based on the same arbitrarily distributed DistGrid. Section **??** discusses DistGrids with user-supplied, arbitrary sequence indices in detail. Here a global index space range from 1 through 20 is decomposed across 4 DEs. There are 4 PETs in this example with 1 DE per PET. Each PET constructs its local `seqIndexList` variable.

```
  do i=1, 5
#ifdef TEST_I8RANGE_on
    seqIndexList(i) = localPet + (i - 1) * petCount + 1 + seqIndexOffset
#else
    seqIndexList(i) = localPet + (i - 1) * petCount + 1
#endif
  enddo
```

This results in the following cyclic distribution scheme:

```
   DE 0 on PET 0: seqIndexList = (/1, 5, 9, 13, 17/)
   DE 1 on PET 1: seqIndexList = (/2, 6, 10, 14, 18/)
   DE 2 on PET 2: seqIndexList = (/3, 7, 11, 15, 19/)
   DE 3 on PET 3: seqIndexList = (/4, 8, 12, 16, 20/)
```

The local `seqIndexList` variables are then used to create a DistGrid with the indicated arbitrary distribution pattern.

```
  distgrid = ESMF_DistGridCreate(arbSeqIndexList=seqIndexList, rc=rc)
```

The resulting DistGrid is one-dimensional, although the user code may interpret the sequence indices as a 1D map into a problem of higher dimensionality.

In this example the local DE on each PET is associated with a 5 element exclusive region. Providing `seqIndexList` of different size on the different PETs is supported and would result in different number of exclusive elements on each PET.

Halo for a 1D Array from existing memory allocation, created on the 1D arbitrary DistGrid.

Creating an ESMF Array on top of a DistGrid with arbitrary sequence indices is in principle no different from creating an Array on a regular DistGrid. However, while an Array that was created on a regular DistGrid automatically inherits the index space topology information that is contained within the DistGrid object, there is no such topology information available for DistGrid objects with arbitrary sequence indices. As a consequence of this, Arrays created on arbitrary DistGrids do not automatically have the information that is required to associated halo elements with the exclusive elements across DEs. Instead the user must supply this information explicitly during Array creation.

Multiple ArrayCreate() interfaces exist that allow the creation of an Array on a DistGrid with arbitrary sequence indices. The sequence indices for the halo region of the local DE are supplied through an additional argument with dummy name `haloSeqIndexList`. As in the regular case, the ArrayCreate() interfaces differ in the way that the memory allocations for the Array elements are passed into the call. The following code shows how an ESMF Array can be wrapped around existing PET-local memory allocations. The allocations are of different size on each PET as to accommodate the correct number of local Array elements (exclusive region + halo region).

```
  allocate(farrayPtr1d(5+localPet+1)) !use explicit Fortran allocate statement

  if (localPet==0) then
    allocate(haloList(1))
#ifdef TEST_I8RANGE_on
    haloList(:)=(/1099511627782_ESMF_KIND_I8/)
#else
    haloList(:)=(/6/)
#endif
    array = ESMF_ArrayCreate(distgrid, farrayPtr1d, &
      haloSeqIndexList=haloList, rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  endif
  if (localPet==1) then
    allocate(haloList(2))
#ifdef TEST_I8RANGE_on
    haloList(:)=(/1099511627777_ESMF_KIND_I8,&
                  1099511627795_ESMF_KIND_I8/)
#else
    haloList(:)=(/1,19/)
#endif
    array = ESMF_ArrayCreate(distgrid, farrayPtr1d, &
      haloSeqIndexList=haloList, rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  endif
  if (localPet==2) then
    allocate(haloList(3))
#ifdef TEST_I8RANGE_on
    haloList(:)=(/1099511627792_ESMF_KIND_I8,&
                  1099511627782_ESMF_KIND_I8,&
                  1099511627785_ESMF_KIND_I8/)
#else
    haloList(:)=(/16,6,9/)
#endif
    array = ESMF_ArrayCreate(distgrid, farrayPtr1d, &
      haloSeqIndexList=haloList, rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  endif
  if (localPet==3) then
    allocate(haloList(4))
#ifdef TEST_I8RANGE_on
    haloList(:)=(/1099511627777_ESMF_KIND_I8,&
                  1099511627779_ESMF_KIND_I8,&
                  1099511627777_ESMF_KIND_I8,&
                  1099511627780_ESMF_KIND_I8/)
#else
    haloList(:)=(/1,3,1,4/)
#endif
    array = ESMF_ArrayCreate(distgrid, farrayPtr1d, &
      haloSeqIndexList=haloList, rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  endif
```

The `haloSeqIndexList` arguments are 1D arrays of sequence indices. It is through this argument that the user

associates the halo elements with exclusive Array elements covered by the DistGrid. In this example there are different number of halo elements on each DE. They are associated with exclusive elements as follows:

```
halo on DE 0 on PET 0: <seqIndex=6>  2nd exclusive element on DE 1
halo on DE 1 on PET 1: <seqIndex=1>  1st exclusive element on DE 0
                       <seqIndex=19> 5th exclusive element on DE 2
halo on DE 2 on PET 2: <seqIndex=16> 4th exclusive element on DE 3
                       <seqIndex=6>  2nd exclusive element on DE 1
                       <seqIndex=9>  3rd exclusive element on DE 0
halo on DE 3 on PET 3: <seqIndex=1>  1st exclusive element on DE 0
                       <seqIndex=3>  1st exclusive element on DE 2
                       <seqIndex=1>  1st exclusive element on DE 0
                       <seqIndex=4>  1st exclusive element on DE 3
```

The above `haloSeqIndexList` arguments were constructed very artificially in order to show the following general features:

- There is no restriction on the order in which the indices in a `haloSeqIndexList` can appear.
- The same sequence index may appear in multiple `haloSeqIndexList` arguments.
- The same sequence index may appear multiple times in the same `haloSeqIndexList` argument.
- A local sequence index may appear in a `haloSeqIndexList` argument.

The ArrayCreate() call checks that the provided Fortran memory allocation is correctly sized to hold the exclusive elements, as indicated by the DistGrid object, plus the halo elements as indicated by the local `haloSeqIndexList` argument. The size of the Fortran allocation must match exactly or a runtime error will be returned.

Analogous to the case of Arrays on regular DistGrids, it is the exclusive region of the local DE that is typically modified by the code running on each PET. All of the ArrayCreate() calls that accept the `haloSeqIndexList` argument place the exclusive region at the beginning of the memory allocation on each DE and use the remaining space for the halo elements. The following loop demonstrates this by filling the exclusive elements on each DE with initial values. Remember that in this example each DE holds 5 exclusive elements associated with different arbitrary sequence indices.

```
farrayPtr1d = 0 ! initialize
do i=1, 5
  farrayPtr1d(i) = real(seqIndexList(i), ESMF_KIND_R8)
enddo
print *, "farrayPtr1d: ", farrayPtr1d
```

Now the exclusive elements of `array` are initialized on each DE, however, the halo elements remain unchanged. A RouteHandle can be set up that encodes the required communication pattern for a halo exchange. The halo exchange is precomputed according to the arbitrary sequence indices specified for the exclusive elements by the DistGrid and the sequence indices provided by the user for each halo element on the local DE in form of the `haloSeqIndexList` argument during ArrayCreate().

```
call ESMF_ArrayHaloStore(array, routehandle=haloHandle, rc=rc)
```

Executing this halo operation will update the local halo elements according to the associated sequence indices.

```
call ESMF_ArrayHalo(array, routehandle=haloHandle, rc=rc)
```

As always it is good practice to release the RouteHandle when done with it.

```
call ESMF_ArrayHaloRelease(haloHandle, rc=rc)
```

Also the Array object should be destroyed when no longer needed.

```
call ESMF_ArrayDestroy(array, rc=rc)
```

Further, since the memory allocation was done explicitly using the Fortran `allocate()` statement, it is necessary to explicitly deallocate in order to prevent memory leaks in the user application.

```
deallocate(farrayPtr1d)
```

Halo for a 1D Array with ESMF managed memory allocation, created on the 1D arbitrary DistGrid.

Alternatively the exact same Array can be created where ESMF does the memory allocation and deallocation. In this case the `typekind` of the Array must be specified explicitly.

```
if (localPet==0) then
  array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
    haloSeqIndexList=haloList, rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif
if (localPet==1) then
  array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
    haloSeqIndexList=haloList, rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif
if (localPet==2) then
  array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
    haloSeqIndexList=haloList, rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif
if (localPet==3) then
  array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
    haloSeqIndexList=haloList, rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif
```

Use `ESMF_ArrayGet()` to gain access to the local memory allocation.

```
call ESMF_ArrayGet(array, farrayPtr=farrayPtr1d, rc=rc)
```

The returned Fortran pointer can now be used to initialize the exclusive elements on each DE as in the previous case.

```
do i=1, 5
  farrayPtr1d(i) = real(seqIndexList(i),ESMF_KIND_R8) / 10.d0
enddo
```

Identical halo operations are constructed and used.

```
call ESMF_ArrayHaloStore(array, routehandle=haloHandle, rc=rc)
```

```
call ESMF_ArrayHalo(array, routehandle=haloHandle, rc=rc)
```

```
call ESMF_ArrayHaloRelease(haloHandle, rc=rc)
```

```
call ESMF_ArrayDestroy(array, rc=rc)
```

Halo for an Array with undistributed dimensions, created on the 1D arbitrary DistGrid, with default Array to DistGrid dimension mapping.

A current limitation of the Array implementation restricts DistGrids that contain user-specified, arbitrary sequence indices to be exactly 1D when used to create Arrays. See section 28.3 for a list of current implementation restrictions. However, an Array created on such a 1D arbitrary DistGrid is allowed to have undistributed dimensions. The following example creates an Array on the same arbitrary DistGrid, with the same arbitrary sequence indices for the halo elements as before, but with one undistributed dimension with a size of 3.

```
if (localPet==0) then
  array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
    haloSeqIndexList=haloList, undistLBound=(/1/), undistUBound=(/3/), rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif
if (localPet==1) then
  array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
    haloSeqIndexList=haloList, undistLBound=(/1/), undistUBound=(/3/), rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif
if (localPet==2) then
  array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
    haloSeqIndexList=haloList, undistLBound=(/1/), undistUBound=(/3/), rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif
if (localPet==3) then
  array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
    haloSeqIndexList=haloList, undistLBound=(/1/), undistUBound=(/3/), rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif
```

By default the DistGrid dimension is mapped to the first Array dimension, associating the remaining Array dimensions with the undistributed dimensions in sequence. The dimension order is important when accessing the individual Array elements. Here the same initialization as before is extended to cover the undistributed dimension.

```
call ESMF_ArrayGet(array, farrayPtr=farrayPtr2d, rc=rc)


do j=1, 3
  do i=1, 5
    farrayPtr2d(i,j) = real(seqIndexList(i),ESMF_KIND_R8) / 10.d0 + 100.d0*j
  enddo
enddo
```

In the context of the Array halo operation additional undistributed dimensions are treated in a simple factorized manner. The same halo association between elements that is encoded in the 1D arbitrary sequence index scheme is applied to each undistributed element separately. This is completely transparent on the user level and the same halo methods are used as before.

```
call ESMF_ArrayHaloStore(array, routehandle=haloHandle, rc=rc)


call ESMF_ArrayHalo(array, routehandle=haloHandle, rc=rc)


call ESMF_ArrayHaloRelease(haloHandle, rc=rc)


call ESMF_ArrayDestroy(array, rc=rc)
```

Halo for an Array with undistributed dimensions, created on the 1D arbitrary DistGrid, mapping the undistributed dimension first.

In some situations it is more convenient to associate some or all of the undistributed dimensions with the first Array dimensions. This can be done easily by explicitly mapping the DistGrid dimension to an Array dimension other than the first one. The `distgridToArrayMap` argument is used to provide this information. The following code creates essentially the same Array as before, but with swapped dimension order – now the first Array dimension is the undistributed one.

```
if (localPet==0) then
  array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
    distgridToArrayMap=(/2/), haloSeqIndexList=haloList, &
    undistLBound=(/1/), undistUBound=(/3/), rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif
if (localPet==1) then
  array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
    distgridToArrayMap=(/2/), haloSeqIndexList=haloList, &
    undistLBound=(/1/), undistUBound=(/3/), rc=rc)
```

```
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  endif
  if (localPet==2) then
    array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
      distgridToArrayMap=(/2/), haloSeqIndexList=haloList, &
      undistLBound=(/1/), undistUBound=(/3/), rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  endif
  if (localPet==3) then
#ifdef TEST_I8RANGE_on
    haloList(:)=(/1099511627777_ESMF_KIND_I8,&
                  1099511627780_ESMF_KIND_I8,&
                  1099511627779_ESMF_KIND_I8,&
                  1099511627778_ESMF_KIND_I8/)
#else
    haloList(:)=(/1,3,5,4/)
#endif
    array = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
      distgridToArrayMap=(/2/), haloSeqIndexList=haloList, &
      undistLBound=(/1/), undistUBound=(/3/), rc=rc)
    if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
  endif
```

Notice that the `haloList` constructed on PET 3 is different from the previous examples. All other PETs reuse the same `haloList` as before. In the previous examples the list loaded into PET 3's `haloSeqIndexList` argument contained a duplicate sequence index. However, now that the undistributed dimension is placed first, the `ESMF_ArrayHaloStore()` call will try to optimize the data exchange by vectorizing it. Duplicate sequence indices are currently *not* supported during vectorization.

When accessing the Array elements, the swapped dimension order results in a swapping of `i` and `j`. This can be seen in the following initialization loop.

```
  call ESMF_ArrayGet(array, farrayPtr=farrayPtr2d, rc=rc)


  do j=1, 3
    do i=1, 5
      farrayPtr2d(j,i) = real(seqIndexList(i),ESMF_KIND_R8) / 10.d0 + 100.d0*j
    enddo
  enddo
```

Once set up, there is no difference in how the the halo operations are applied.

```
  call ESMF_ArrayHaloStore(array, routehandle=haloHandle, rc=rc)


  call ESMF_ArrayHalo(array, routehandle=haloHandle, rc=rc)


  call ESMF_ArrayDestroy(array, rc=rc)
```

Halo for an Array with undistributed dimensions, created on the 1D arbitrary DistGrid, re-using the RouteHandle.

Arrays can reuse the same RouteHandle, saving the overhead that is caused by the precompute step. In order to demonstrate this the RouteHandle of the previous halo call was not yet released and will be applied to a new Array.

The following code creates an Array that is compatible to the previous Array by using the same input information as before, only that the size of the undistributed dimension is now 6 instead of 3.

```
if (localPet==0) then
  array2 = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
    distgridToArrayMap=(/2/), haloSeqIndexList=haloList, &
    undistLBound=(/1/), undistUBound=(/6/), rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif
if (localPet==1) then
  array2 = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
    distgridToArrayMap=(/2/), haloSeqIndexList=haloList, &
    undistLBound=(/1/), undistUBound=(/6/), rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif
if (localPet==2) then
  array2 = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
    distgridToArrayMap=(/2/), haloSeqIndexList=haloList, &
    undistLBound=(/1/), undistUBound=(/6/), rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif
if (localPet==3) then
  array2 = ESMF_ArrayCreate(distgrid=distgrid, typekind=ESMF_TYPEKIND_R8, &
    distgridToArrayMap=(/2/), haloSeqIndexList=haloList, &
    undistLBound=(/1/), undistUBound=(/6/), rc=rc)
  if (rc /= ESMF_SUCCESS) call ESMF_Finalize(endflag=ESMF_END_ABORT)
endif
```

Again the exclusive Array elements must be initialized.

```
call ESMF_ArrayGet(array2, farrayPtr=farrayPtr2d, rc=rc)


do j=1, 6
  do i=1, 5
    farrayPtr2d(j,i) = real(seqIndexList(i),ESMF_KIND_R8) / 10.d0 + 100.d0*j
  enddo
enddo
```

Now the `haloHandle` that was previously pre-computed for `array` can be used directly for `array2`.

```
call ESMF_ArrayHalo(array2, routehandle=haloHandle, rc=rc)
```

Release the RouteHandle after its last use and clean up the remaining Array and DistGrid objects.

```
call ESMF_ArrayHaloRelease(haloHandle, rc=rc)


call ESMF_ArrayDestroy(array2, rc=rc)


call ESMF_DistGridDestroy(distgrid, rc=rc)
```

28.2.17 Communication – Redist

Arrays used in different models often cover the same index space region, however, the distribution of the Arrays may be different, e.g. the models run on exclusive sets of PETs. Even if the Arrays are defined on the same list of PETs the decomposition may be different.

```
srcDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/10,20/), &
  regDecomp=(/4,1/), rc=rc)


dstDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/10,20/), &
  regDecomp=(/1,4/), rc=rc)
```

The number of elements covered by `srcDistgrid` is identical to the number of elements covered by `dstDistgrid` – in fact the index space regions covered by both DistGrid objects are congruent. However, the decomposition defined by `regDecomp`, and consequently the distribution of source and destination, are different.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)


srcArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=srcDistgrid, rc=rc)


dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, rc=rc)
```

By construction `srcArray` and `dstArray` are of identical type and kind. Further the number of exclusive elements matches between both Arrays. These are the prerequisites for the application of an Array redistribution in default mode. In order to increase performance of the actual redistribution the communication pattern is precomputed once, and stored in an `ESMF_RouteHandle` object.

```
call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, rc=rc)
```

The `redistHandle` can now be used repeatedly to transfer data from `srcArray` to `dstArray`.

```
call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, rc=rc)
```

The use of the precomputed `redistHandle` is *not* restricted to the (`srcArray`, `dstArray`) pair. Instead the `redistHandle` can be used to redistribute data between any two Arrays that are compatible with the Array pair used during precomputation. I.e. any pair of Arrays that matches `srcArray` and `dstArray` in *type*, *kind*, and memory layout of the *distributed* dimensions. However, the size, number, and index order of *undistributed* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

The transferability of RouteHandles between Array pairs can greatly reduce the number of communication store calls needed. In a typical application Arrays are often defined on the same decomposition, typically leading to congruent distributed dimensions. For these Arrays, while they may not have the same shape or size in the undistributed dimensions, RouteHandles are reusable.

For the current case, the `redistHandle` was precomputed for simple 2D Arrays without undistributed dimensions. The RouteHandle transferability rule allows us to use this same RouteHandle to redistribute between two 3D Array that are built on the same 2D DistGrid, but have an undistributed dimension. Note that the undistributed dimension does not have to be in the same position on source and destination. Here the undistributed dimension is in position 2 for `srcArray1`, and in position 1 for `dstArray1`.

```
call ESMF_ArraySpecSet(arrayspec3d, typekind=ESMF_TYPEKIND_R8, rank=3, rc=rc)


srcArray1 = ESMF_ArrayCreate(arrayspec=arrayspec3d, distgrid=srcDistgrid, &
  distgridToArrayMap=(/1,3/), undistLBound=(/1/), undistUBound=(/10/), rc=rc)


dstArray1 = ESMF_ArrayCreate(arrayspec=arrayspec3d, distgrid=dstDistgrid, &
  distgridToArrayMap=(/2,3/), undistLBound=(/1/), undistUBound=(/10/), rc=rc)


call ESMF_ArrayRedist(srcArray=srcArray1, dstArray=dstArray1, &
  routehandle=redistHandle, rc=rc)
```

The following variation of the code shows that the same RouteHandle can be applied to an Array pair where the number of undistributed dimensions does not match between source and destination Array. Here we prepare a source Array with *two* undistributed dimensions, in position 1 and 3, that multiply out to 2x5=10 undistributed elements. The destination array is the same as before with only a *single* undistributed dimension in position 1 of size 10.

```
call ESMF_ArraySpecSet(arrayspec4d, typekind=ESMF_TYPEKIND_R8, rank=4, rc=rc)


srcArray2 = ESMF_ArrayCreate(arrayspec=arrayspec4d, distgrid=srcDistgrid, &
  distgridToArrayMap=(/2,4/), undistLBound=(/1,1/), undistUBound=(/2,5/), &
  rc=rc)


call ESMF_ArrayRedist(srcArray=srcArray2, dstArray=dstArray1, &
  routehandle=redistHandle, rc=rc)
```

When done, the resources held by `redistHandle` need to be deallocated by the user code before the RouteHandle becomes inaccessible.

```
call ESMF_ArrayRedistRelease(routehandle=redistHandle, rc=rc)
```

In *default* mode, i.e. without providing the optional `srcToDstTransposeMap` argument, `ESMF_ArrayRedistStore()` does not require equal number of dimensions in source and destination Array. Only the total number of elements must match. Specifying `srcToDstTransposeMap` switches `ESMF_ArrayRedistStore()` into *transpose* mode. In this mode each dimension of `srcArray` is uniquely associated with a dimension in `dstArray`, and the sizes of associated dimensions must match for each pair.

```
dstDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/20,10/), &
    rc=rc)


dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, rc=rc)
```

This `dstArray` object covers a 20 x 10 index space while the `srcArray`, defined further up, covers a 10 x 20 index space. Setting `srcToDstTransposeMap = (/2,1/)` will associate the first and second dimension of `srcArray` with the second and first dimension of `dstArray`, respectively. This corresponds to a transpose of dimensions. Since the decomposition and distribution of dimensions may be different for source and destination redistribution may occur at the same time.

```
call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, srcToDstTransposeMap=(/2,1/), rc=rc)


call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, rc=rc)
```

The transpose mode of `ESMF_ArrayRedist()` is not limited to distributed dimensions of Arrays. The `srcToDstTransposeMap` argument can be used to transpose undistributed dimensions in the same manner. Furthermore transposing distributed and undistributed dimensions between Arrays is also supported.

The `srcArray` used in the following examples is of rank 4 with 2 distributed and 2 undistributed dimensions. The distributed dimensions are the two first dimensions of the Array and are distributed according to the `srcDistgrid` which describes a total index space region of 100 x 200 elements. The last two Array dimensions are undistributed dimensions of size 2 and 3, respectively.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=4, rc=rc)


srcDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/100,200/), &
  rc=rc)


srcArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=srcDistgrid, &
  undistLBound=(/1,1/), undistUBound=(/2,3/), rc=rc)
```

The first `dstArray` to consider is defined on a DistGrid that also describes a 100 x 200 index space region. The distribution indicated by `dstDistgrid` may be different from the source distribution. Again the first two Array dimensions are associated with the DistGrid dimensions in sequence. Furthermore, the last two Array dimensions are undistributed dimensions, however, the sizes are 3 and 2, respectively.

```
dstDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/100,200/), &
  rc=rc)


dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, &
  undistLBound=(/1,1/), undistUBound=(/3,2/), rc=rc)
```

The desired mapping between `srcArray` and `dstArray` dimensions is expressed by `srcToDstTransposeMap = (/1,2,4,3/)`, transposing only the two undistributed dimensions.

```
call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, srcToDstTransposeMap=(/1,2,4,3/), rc=rc)


call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, rc=rc)
```

Next consider a `dstArray` that is defined on the same `dstDistgrid`, but with a different order of Array dimensions. The desired order is specified during Array creation using the argument `distgridToArrayMap = (/2,3/)`. This map associates the first and second DistGrid dimensions with the second and third Array dimensions, respectively, leaving Array dimensions one and four undistributed.

```
dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, &
  distgridToArrayMap=(/2,3/), undistLBound=(/1,1/), undistUBound=(/3,2/), &
  rc=rc)
```

Again the sizes of the undistributed dimensions are chosen in reverse order compared to `srcArray`. The desired transpose mapping in this case will be `srcToDstTransposeMap = (/2,3,4,1/)`.

```
call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, srcToDstTransposeMap=(/2,3,4,1/), rc=rc)


call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, rc=rc)
```

Finally consider the case where `dstArray` is constructed on a 200 x 3 index space and where the undistributed dimensions are of size 100 and 2.

```
dstDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/200,3/), &
  rc=rc)


dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, &
  undistLBound=(/1,1/), undistUBound=(/100,2/), rc=rc)
```

By construction srcArray and dstArray hold the same number of elements, albeit in a very different layout. Nevertheless, with a srcToDstTransposeMap that maps matching dimensions from source to destination an Array redistribution becomes a well defined operation between srcArray and dstArray.

```
call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, srcToDstTransposeMap=(/3,1,4,2/), rc=rc)


call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, rc=rc)
```

The default mode of Array redistribution, i.e. without providing a srcToDstTransposeMap to ESMF_ArrayRedistStore(), also supports undistributed Array dimensions. The requirement in this case is that the total undistributed element count, i.e. the product of the sizes of all undistributed dimensions, be the same for source and destination Array. In this mode the number of undistributed dimensions need not match between source and destination.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=4, rc=rc)


srcDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/10,20/), &
  regDecomp=(/4,1/), rc=rc)


srcArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=srcDistgrid, &
  undistLBound=(/1,1/), undistUBound=(/2,4/), rc=rc)


dstDistgrid = ESMF_DistGridCreate(minIndex=(/1,1/), maxIndex=(/10,20/), &
  regDecomp=(/1,4/), rc=rc)


dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, &
  distgridToArrayMap=(/2,3/), undistLBound=(/1,1/), undistUBound=(/2,4/), &
  rc=rc)
```

Both srcArray and dstArray have two undistributed dimensions and a total count of undistributed elements of $2 \times 4 = 8$.

The Array redistribution operation is defined in terms of sequentialized undistributed dimensions. In the above case this means that a unique sequence index will be assigned to each of the 8 undistributed elements. The sequence indices will be 1, 2, ..., 8, where sequence index 1 is assigned to the first element in the first (i.e. fastest varying in memory) undistributed dimension. The following undistributed elements are labeled in consecutive order as they are stored in memory.

```
call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, rc=rc)
```

The redistribution operation by default applies the identity operation between the elements of undistributed dimensions. This means that source element with sequence index 1 will be mapped against destination element with sequence index 1 and so forth. Because of the way source and destination Arrays in the current example were constructed this corresponds to a mapping of dimensions 3 and 4 on `srcArray` to dimensions 1 and 4 on `dstArray`, respectively.

```
call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, rc=rc)
```

Array redistribution does *not* require the same number of undistributed dimensions in source and destination Array, merely the total number of undistributed elements must match.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=3, rc=rc)


dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, &
  distgridToArrayMap=(/1,3/), undistLBound=(/11/), undistUBound=(/18/), &
  rc=rc)
```

This `dstArray` object only has a single undistributed dimension, while the `srcArray`, defined further back, has two undistributed dimensions. However, the total undistributed element count for both Arrays is 8.

```
call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, rc=rc)
```

In this case the default identity operation between the elements of undistributed dimensions corresponds to a *merging* of dimensions 3 and 4 on `srcArray` into dimension 2 on `dstArray`.

```
call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=redistHandle, rc=rc)
```

28.2.18 Communication – SparseMatMul

Sparse matrix multiplication is a fundamental Array communication method. One frequently used application of this method is the interpolation between pairs of Arrays. The principle is this: the value of each element in the exclusive region of the destination Array is expressed as a linear combination of *potentially all* the exclusive elements of the source Array. Naturally most of the coefficients of these linear combinations will be zero and it is more efficient to store explicit information about the non-zero elements than to keep track of all the coefficients.

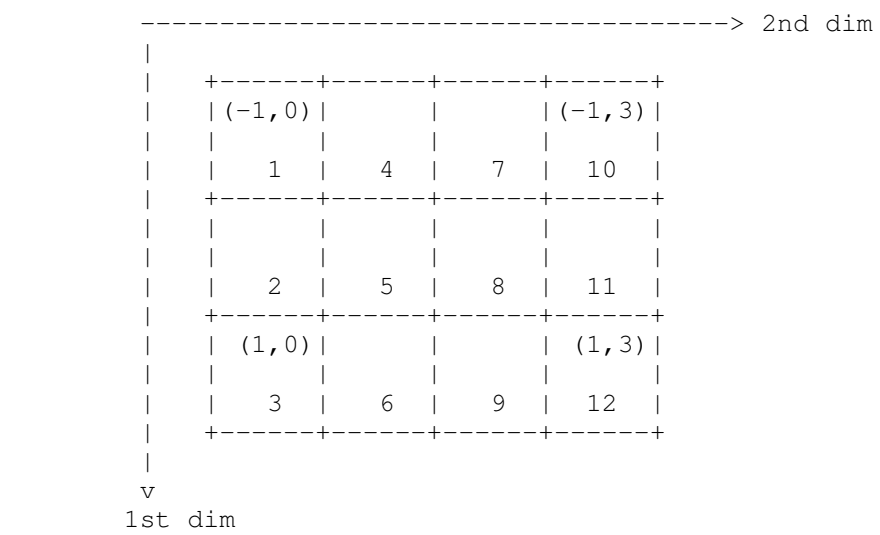
There is a choice to be made with respect to the format in which to store the information about the non-zero elements. One option is to store the value of each coefficient together with the corresponding destination element index and source element index. Destination and source indices could be expressed in terms of the corresponding DistGrid tile index together with the coordinate tuple within the tile. While this format may be the most natural way to express elements in the source and destination Array, it has two major drawbacks. First the coordinate tuple is `dimCount` specific and second the format is extremely bulky. For 2D source and destination Arrays it would require 6 integers to store the source and destination element information for each non-zero coefficient and matters get worse for higher dimensions.

Both problems can be circumvented by *interpreting* source and destination Arrays as sequentialized strings or *vectors* of elements. This is done by assigning a unique *sequence index* to each exclusive element in both Arrays. With that the operation of updating the elements in the destination Array as linear combinations of source Array elements takes the form of a *sparse matrix multiplication*.

The default sequence index rule assigns index 1 to the `minIndex` corner element of the first tile of the DistGrid on which the Array is defined. It then increments the sequence index by 1 for each element running through the DistGrid dimensions by order. The index space position of the DistGrid tiles does not affect the sequence labeling of elements. The default sequence indices for

```
srcDistgrid = ESMF_DistGridCreate(minIndex=(/-1,0/), maxIndex=(/1,3/), rc=rc)
```

for each element are:



The assigned sequence indices are decomposition and distribution invariant by construction. Furthermore, when an Array is created with extra elements per DE on a DistGrid the sequence indices (which only cover the exclusive elements) remain unchanged.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)
```

```
srcArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=srcDistgrid, &
  totalLWidth=(/1,1/), totalUWidth=(/1,1/), indexflag=ESMF_INDEX_GLOBAL, &
  rc=rc)
```

The extra padding of 1 element in each direction around the exclusive elements on each DE are "invisible" to the Array sparse matrix multiplication method. These extra elements are either updated by the computational kernel or by Array halo operations.

An alternative way to assign sequence indices to all the elements in the tiles covered by a DistGrid object is to use a special `ESMF_DistGridCreate()` call. This call has been specifically designed for 1D cases with arbitrary, user-supplied sequence indices.

```
seqIndexList(1) = localPet*10
seqIndexList(2) = localPet*10 + 1
dstDistgrid = ESMF_DistGridCreate(arbSeqIndexList=seqIndexList, rc=rc)
```

This call to `ESMF_DistGridCreate()` is collective across the current VM. The `arbSeqIndexList` argument specifies the PET-local arbitrary sequence indices that need to be covered by the local DE. The resulting DistGrid has one local DE per PET which covers the entire PET-local index range. The user supplied sequence indices must be unique, but the sequence may be interrupted. The four DEs of `dstDistgrid` have the following local 1D index space coordinates (given between "()") and sequence indices:

```
covered by DE 0    covered by DE 1   covered by DE 2   covered by DE 3
on PET 0           on PET 1          on PET 2          on PET 3
-------------------------------------------------------------------
(1) : 0            (1) : 10          (1) : 20          (1) : 30
(2) : 1            (2) : 11          (2) : 21          (2) : 31
```

Again the DistGrid object provides the sequence index labeling for the exclusive elements of an Array created on the DistGrid regardless of extra, non-exclusive elements.

```
dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, rc=rc)
```

With the definition of sequence indices, either by the default rule or as user provided arbitrary sequence indices, it is now possible to uniquely identify each exclusive element in the source and destination Array by a single integer number. Specifying a pair of source and destination elements takes two integer number regardless of the number of dimensions.

The information required to carry out a sparse matrix multiplication are the pair of source and destination sequence indices and the associated multiplication factor for each pair. ESMF requires this information in form of two Fortran arrays. The factors are stored in a 1D array of the appropriate type and kind, e.g. `real(ESMF_KIND_R8)::factorList(:)`. Array sparse matrix multiplications are supported between Arrays of different type and kind. The type and kind of the factors can also be chosen freely. The sequence index pairs associated with the factors provided by `factorList` are stored in a 2D Fortran array of default integer kind of the shape `integer::factorIndexList(2,:)`. The sequence indices of the source Array elements are stored in the first row of `factorIndexList` while the sequence indices of the destination Array elements are stored in the second row.

Each PET in the current VM must call into `ESMF_ArraySMMStore()` to precompute and store the communication pattern for the sparse matrix multiplication. The multiplication factors may be provided in parallel, i.e. multiple PETs may specify `factorList` and `factorIndexList` arguments when calling into `ESMF_ArraySMMStore()`. PETs that do not provide factors either call with `factorList` and `factorIndexList` arguments containing zero elements or issue the call omitting both arguments.

```
if (localPet == 0) then
  allocate(factorList(1))                ! PET 0 specifies 1 factor
  allocate(factorIndexList(2,1))
  factorList = (/0.2/)                   ! factors
  factorIndexList(1,:) = (/5/)           ! seq indices into srcArray
  factorIndexList(2,:) = (/30/)          ! seq indices into dstArray

  call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
    routehandle=sparseMatMulHandle, factorList=factorList, &
```

```
    factorIndexList=factorIndexList, rc=rc)



  deallocate(factorList)
  deallocate(factorIndexList)
else if (localPet == 1) then
  allocate(factorList(3))               ! PET 1 specifies 3 factor
  allocate(factorIndexList(2,3))
  factorList = (/0.5, 0.5, 0.8/)        ! factors
  factorIndexList(1,:) = (/8, 2, 12/)   ! seq indices into srcArray
  factorIndexList(2,:) = (/11, 11, 30/) ! seq indices into dstArray

  call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
    routehandle=sparseMatMulHandle, factorList=factorList, &
    factorIndexList=factorIndexList, rc=rc)



  deallocate(factorList)
  deallocate(factorIndexList)
else
  ! PETs 2 and 3 do not provide factors

  call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
    routehandle=sparseMatMulHandle, rc=rc)



endif
```

The RouteHandle object `sparseMatMulHandle` produced by `ESMF_ArraySMMStore()` can now be used to call `ESMF_ArraySMM()` collectively across all PETs of the current VM to perform

```
    dstArray = 0.0
    do n=1, size(combinedFactorList)
        dstArray(combinedFactorIndexList(2, n)) +=
          combinedFactorList(n) * srcArray(combinedFactorIndexList(1, n))
    enddo
```

in parallel. Here `combinedFactorList` and `combinedFactorIndexList` are the combined lists defined by the respective local lists provided by PETs 0 and 1 in parallel. For this example

```
call ESMF_ArraySMM(srcArray=srcArray, dstArray=dstArray, &
  routehandle=sparseMatMulHandle, rc=rc)
```

will initialize the entire `dstArray` to 0.0 and then update two elements:

```
 on DE 1:
 dstArray(2) = 0.5 * srcArray(0,0)  +  0.5 * srcArray(0,2)
```

and

```
on DE 3:
dstArray(1) = 0.2 * srcArray(0,1)  +  0.8 * srcArray(1,3).
```

The call to `ESMF_ArraySMM()` does provide the option to turn the default `dstArray` initialization off. If argument `zeroregion` is set to `ESMF_REGION_EMPTY`

```
call ESMF_ArraySMM(srcArray=srcArray, dstArray=dstArray, &
  routehandle=sparseMatMulHandle, zeroregion=ESMF_REGION_EMPTY, rc=rc)
```

skips the initialization and elements in `dstArray` are updated according to:

```
do n=1, size(combinedFactorList)
    dstArray(combinedFactorIndexList(2, n)) +=
      combinedFactorList(n) * srcArray(combinedFactorIndexList(1, n)).
enddo
```

The `ESMF_RouteHandle` object returned by `ESMF_ArraySMMStore()` can be applied to any src/dst Array pairs that is compatible with the Array pair used during precomputation, i.e. any pair of Arrays that matches `srcArray` and `dstArray` in *type*, *kind*, and memory layout of the *distributed* dimensions. However, the size, number, and index order of *undistributed* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

The resources held by `sparseMatMulHandle` need to be deallocated by the user code before the handle becomes inaccessible.

```
call ESMF_ArraySMMRelease(routehandle=sparseMatMulHandle, rc=rc)
```

The Array sparse matrix multiplication also applies to Arrays with undistributed dimensions. The undistributed dimensions are interpreted in a sequentialized manner, much like the distributed dimensions, introducing a second sequence index for source and destination elements. Sequence index 1 is assigned to the first element in the first (i.e. fastest varying in memory) undistributed dimension. The following undistributed elements are labeled in consecutive order as they are stored in memory.

In the simplest case the Array sparse matrix multiplication will apply an identity matrix to the vector of sequentialized undistributed Array elements for every non-zero element in the sparse matrix. The requirement in this case is that the total undistributed element count, i.e. the product of the sizes of all undistributed dimensions, be the same for source and destination Array.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=3, rc=rc)
srcArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=srcDistgrid, &
  totalLWidth=(/1,1/), totalUWidth=(/1,1/), indexflag=ESMF_INDEX_GLOBAL, &
  distgridToArrayMap=(/1,2/), undistLBound=(/1/), undistUBound=(/2/), rc=rc)


call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)
dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, &
  distgridToArrayMap=(/2/), undistLBound=(/1/), undistUBound=(/2/), rc=rc)
```

Setting up `factorList` and `factorIndexList` is identical to the case for Arrays without undistributed dimensions. Also the call to `ESMF_ArraySMMStore()` remains unchanged. Internally, however, the source and destination Arrays are checked to make sure the total undistributed element count matches.

```
if (localPet == 0) then
  allocate(factorList(1))               ! PET 0 specifies 1 factor
  allocate(factorIndexList(2,1))
  factorList = (/0.2/)                  ! factors
  factorIndexList(1,:) = (/5/)          ! seq indices into srcArray
  factorIndexList(2,:) = (/30/)         ! seq indices into dstArray

  call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
    routehandle=sparseMatMulHandle, factorList=factorList, &
    factorIndexList=factorIndexList, rc=rc)



  deallocate(factorList)
  deallocate(factorIndexList)
else if (localPet == 1) then
  allocate(factorList(3))               ! PET 1 specifies 3 factor
  allocate(factorIndexList(2,3))
  factorList = (/0.5, 0.5, 0.8/)        ! factors
  factorIndexList(1,:) = (/8, 2, 12/)   ! seq indices into srcArray
  factorIndexList(2,:) = (/11, 11, 30/) ! seq indices into dstArray

  call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
    routehandle=sparseMatMulHandle, factorList=factorList, &
    factorIndexList=factorIndexList, rc=rc)




  deallocate(factorList)
  deallocate(factorIndexList)
else
  ! PETs 2 and 3 do not provide factors

  call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
    routehandle=sparseMatMulHandle, rc=rc)



endif
```

The call into the `ESMF_ArraySMM()` operation is completely transparent with respect to whether source and/or destination Arrays contain undistributed dimensions.

```
call ESMF_ArraySMM(srcArray=srcArray, dstArray=dstArray, &
  routehandle=sparseMatMulHandle, rc=rc)
```

This operation will initialize the entire `dstArray` to 0.0 and then update four elements:

```
on DE 1:
dstArray[1](2) = 0.5 * srcArray(0,0)[1]  +  0.5 * srcArray(0,2)[1],
dstArray[2](2) = 0.5 * srcArray(0,0)[2]  +  0.5 * srcArray(0,2)[2]
```

and

```
on DE 3:
dstArray[1](1) = 0.2 * srcArray(0,1)[1]  +  0.8 * srcArray(1,3)[1],
dstArray[2](1) = 0.2 * srcArray(0,1)[2]  +  0.8 * srcArray(1,3)[2].
```

Here indices between "()" refer to distributed dimensions while indices between "[]" correspond to undistributed dimensions.

In a more general version of the Array sparse matrix multiplication the total undistributed element count, i.e. the product of the sizes of all undistributed dimensions, need not be the same for source and destination Array. In this formulation each non-zero element of the sparse matrix is identified with a unique element in the source and destination Array. This requires a generalization of the `factorIndexList` argument which now must contain four integer numbers for each element. These numbers in sequence are the sequence index of the distributed dimensions and the sequence index of the undistributed dimensions of the element in the source Array, followed by the sequence index of the distributed dimensions and the sequence index of the undistributed dimensions of the element in the destination Array.

```
call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=3, rc=rc)
srcArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=srcDistgrid, &
  totalLWidth=(/1,1/), totalUWidth=(/1,1/), indexflag=ESMF_INDEX_GLOBAL, &
  distgridToArrayMap=(/1,2/), undistLBound=(/1/), undistUBound=(/2/), rc=rc)


call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_R8, rank=2, rc=rc)
dstArray = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=dstDistgrid, &
  distgridToArrayMap=(/2/), undistLBound=(/1/), undistUBound=(/4/), rc=rc)
```

Setting up `factorList` is identical to the previous cases since there is still only one value associated with each non-zero matrix element. However, each entry in `factorIndexList` now has 4 instead of just 2 components.

```
if (localPet == 0) then
  allocate(factorList(1))               ! PET 0 specifies 1 factor
  allocate(factorIndexList(4,1))
  factorList = (/0.2/)                  ! factors
  factorIndexList(1,:) = (/5/)          ! seq indices into srcArray
  factorIndexList(2,:) = (/1/)          ! undistr. seq indices into srcArray
  factorIndexList(3,:) = (/30/)         ! seq indices into dstArray
  factorIndexList(4,:) = (/2/)          ! undistr. seq indices into dstArray

  call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
    routehandle=sparseMatMulHandle, factorList=factorList, &
    factorIndexList=factorIndexList, rc=rc)
```

```
    deallocate(factorList)
    deallocate(factorIndexList)
  else if (localPet == 1) then
    allocate(factorList(3))               ! PET 1 specifies 3 factor
    allocate(factorIndexList(4,3))
    factorList = (/0.5, 0.5, 0.8/)        ! factors
    factorIndexList(1,:) = (/8, 2, 12/)   ! seq indices into srcArray
    factorIndexList(2,:) = (/2, 1, 1/)    ! undistr. seq indices into srcArray
    factorIndexList(3,:) = (/11, 11, 30/) ! seq indices into dstArray
    factorIndexList(4,:) = (/4, 4, 2/)    ! undistr. seq indices into dstArray

    call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
      routehandle=sparseMatMulHandle, factorList=factorList, &
      factorIndexList=factorIndexList, rc=rc)




    deallocate(factorList)
    deallocate(factorIndexList)
  else
    ! PETs 2 and 3 do not provide factors

    call ESMF_ArraySMMStore(srcArray=srcArray, dstArray=dstArray, &
      routehandle=sparseMatMulHandle, rc=rc)



  endif
```

The call into the `ESMF_ArraySMM()` operation remains unchanged.

```
  call ESMF_ArraySMM(srcArray=srcArray, dstArray=dstArray, &
    routehandle=sparseMatMulHandle, rc=rc)
```

This operation will initialize the entire `dstArray` to 0.0 and then update two elements:

```
   on DE 1:
   dstArray[4](2) = 0.5 * srcArray(0,0)[1]  +  0.5 * srcArray(0,2)[2],
```

and

```
   on DE 3:
   dstArray[2](1) = 0.2 * srcArray(0,1)[1]  +  0.8 * srcArray(1,3)[1],
```

Here indices in () refer to distributed dimensions while indices in [] correspond to undistributed dimensions.

28.2.19 Communication – Scatter and Gather, revisited

The ESMF_ArrayScatter() and ESMF_ArrayGather() calls, introduced in section 28.2.14, provide a convenient way of communicating data between a Fortran array and all of the DEs of a single Array tile. A key requirement of ESMF_ArrayScatter() and ESMF_ArrayGather() is that the *shape* of the Fortran array and the Array tile must match. This means that the dimCount must be equal, and that the size of each dimension must match. Element reordering during scatter and gather is only supported on a per dimension level, based on the decompflag option available during DistGrid creation.

While the ESMF_ArrayScatter() and ESMF_ArrayGather() methods cover a broad, and important spectrum of cases, there are situations that require a different set of rules to scatter and gather data between a Fortran array and an ESMF Array object. For instance, it is often convenient to create an Array on a DistGrid that was created with arbitrary, user-supplied sequence indices. See section **??** for more background on DistGrids with arbitrary sequence indices.

```
allocate(arbSeqIndexList(10))   ! each PET will have 10 elements

do i=1, 10
  arbSeqIndexList(i) = (i-1)*petCount + localPet+1 ! initialize unique
                                                   ! seq. indices
enddo

distgrid = ESMF_DistGridCreate(arbSeqIndexList=arbSeqIndexList, rc=rc)


deallocate(arbSeqIndexList)

call ESMF_ArraySpecSet(arrayspec, typekind=ESMF_TYPEKIND_I4, rank=1, rc=rc)


array = ESMF_ArrayCreate(arrayspec=arrayspec, distgrid=distgrid, rc=rc)
```

This array object holds 10 elements on each DE, and there is one DE per PET, for a total element count of 10 x petCount. The arbSeqIndexList, used during DistGrid creation, was constructed cyclic across all DEs. DE 0, for example, on a 4 PET run, would hold sequence indices 1, 5, 9, ... DE 1 would hold 2, 6, 10, ..., and so on.

The usefulness of the user-specified arbitrary sequence indices becomes clear when they are interpreted as global element ids. The ArrayRedist() and ArraySMM() communication methods are based on sequence index mapping between source and destination Arrays. Other than providing a canonical sequence index order via the default sequence scheme, outlined in 28.2.18, ESMF does not place any restrictions on the sequence indices. Objects that were not created with user supplied sequence indices default to the ESMF sequence index order.

A common, and useful interpretation of the arbitrary sequence indices, specified during DistGrid creation, is that of relating them to the canonical ESMF sequence index order of another data object. Within this interpretation the array object created above could be viewed as an arbitrary distribution of a (petCount x 10) 2D array.

```
if (localPet == 0) then
  allocate(farray(petCount,10)) ! allocate 2D Fortran array petCount x 10
  do j=1, 10
    do i=1, petCount
      farray(i,j) = 100 + (j-1)*petCount + i    ! initialize to something
    enddo
  enddo
```

```
else
  allocate(farray(0,0)) ! must allocate an array of size 0 on all other PETs
endif
```

For a 4 PET run, `farray` on PET 0 now holds the following data.

```
 -----1----2----3------------10-----> j
 |
 1   101, 105, 109, ....  , 137
 |
 2   102, 106, 110, ....  , 138
 |
 3   103, 107, 111, ....  , 139
 |
 4   104, 108, 112, ....  , 140
 |
 |
 v
 i
```

On all other PETs `farray` has a zero size allocation.

Following the sequence index interpretation from above, scattering the data contained in `farray` on PET 0 across the `array` object created further up, seems like a well defined operation. Looking at it a bit closer, it becomes clear that it is in fact more of a redistribution than a simple scatter operation. The general rule for such a "redist-scatter" operation, of a Fortran array, located on a single PET, into an ESMF Array, is to use the canonical ESMF sequence index scheme to label the elements of the Fortran array, and to send the data to the Array element with the same sequence index.

The just described "redist-scatter" operation is much more general than the standard `ESMF_ArrayScatter()` method. It does not require shape matching, and supports full element reordering based on the sequence indices. Before `farray` can be scattered across `array` in the described way, it must be wrapped into an ESMF Array object itself, essentially labeling the array elements according to the canonical sequence index scheme.

```
distgridAux = ESMF_DistGridCreate(minIndex=(/1,1/), &
  maxIndex=(/petCount,10/), &
  regDecomp=(/1,1/), rc=rc) ! DistGrid with only 1 DE
```

The first step is to create a DistGrid object with only a single DE. This DE must be located on the PET on which the Fortran data array resides. In this example `farray` holds data on PET 0, which is where the default DELayout will place the single DE defined in the DistGrid. If the `farray` was setup on a different PET, an explicit DELayout would need to be created first, mapping the only DE to the PET on which the data is defined.

Next the Array wrapper object can be created from the `farray` and the just created DistGrid object.

```
arrayAux = ESMF_ArrayCreate(farray=farray, distgrid=distgridAux, &
  indexflag=ESMF_INDEX_DELOCAL, rc=rc)
```

At this point all of the pieces are in place to use `ESMF_ArrayRedist()` to do the "redist-scatter" operation. The typical store/execute/release pattern must be followed.

```
call ESMF_ArrayRedistStore(srcArray=arrayAux, dstArray=array, &
  routehandle=scatterHandle, rc=rc)


call ESMF_ArrayRedist(srcArray=arrayAux, dstArray=array, &
  routehandle=scatterHandle, rc=rc)
```

In this example, after `ESMF_ArrayRedist()` was called, the content of `array` on a 4 PET run would look like this:

```
  PET 0:   101, 105, 109, ....  , 137
  PET 1:   102, 106, 110, ....  , 138
  PET 2:   103, 107, 111, ....  , 139
  PET 3:   104, 108, 112, ....  , 140
```

Once set up, `scatterHandle` can be used repeatedly to scatter data from `farray` on PET 0 to all the DEs of `array`. All of the resources should be released once `scatterHandle` is no longer needed.

```
call ESMF_ArrayRedistRelease(routehandle=scatterHandle, rc=rc)
```

The opposite operation, i.e. *gathering* of the `array` data into `farray` on PET 0, follows a very similar setup. In fact, the `arrayAux` object already constructed for the scatter direction, can directly be re-used. The only thing that is different for the "redist-gather", are the `srcArray` and `dstArray` argument assignments, reflecting the opposite direction of data movement.

```
call ESMF_ArrayRedistStore(srcArray=array, dstArray=arrayAux, &
  routehandle=gatherHandle, rc=rc)


call ESMF_ArrayRedist(srcArray=array, dstArray=arrayAux, &
  routehandle=gatherHandle, rc=rc)
```

Just as for the scatter case, the `gatherHandle` can be used repeatedly to gather data from `array` into `farray` on PET 0. All of the resources should be released once `gatherHandle` is no longer needed.

```
call ESMF_ArrayRedistRelease(routehandle=gatherHandle, rc=rc)
```

Finally the wrapper Array `arrayAux` and the associated DistGrid object can also be destroyed.

```
call ESMF_ArrayDestroy(arrayAux, rc=rc)


call ESMF_DistGridDestroy(distgridAux, rc=rc)
```

Further, the primary data objects of this example must be deallocated and destroyed.

```
deallocate(farray)

call ESMF_ArrayDestroy(array, rc=rc)


call ESMF_DistGridDestroy(distgrid, rc=rc)
```

28.2.20 Non-blocking Communications

All ESMF_RouteHandle based communication methods, like ESMF_ArrayRedist(), ESMF_ArrayHalo() and ESMF_ArraySMM(), can be executed in blocking or non-blocking mode. The non-blocking feature is useful, for example, to overlap computation with communication, or to implement a more loosely synchronized inter-Component interaction scheme than is possible with the blocking communication mode.

Access to the non-blocking execution mode is provided uniformly across all RouteHandle based communication calls. Every such call contains the optional routesyncflag argument of type ESMF_RouteSync_Flag. Section **??** lists all of the valid settings for this flag.

It is an execution time decision to select whether to invoke a precomputed communication pattern, stored in a RouteHandle, in the blocking or non-blocking mode. Neither requires specifically precomputed RouteHandles - i.e. a RouteHandle is neither specifically blocking nor specifically non-blocking.

```
call ESMF_ArrayRedistStore(srcArray=srcArray, dstArray=dstArray, &
  routehandle=routehandle, rc=rc)
```

The returned RouteHandle routehandle can be used in blocking or non-blocking execution calls. The application is free to switch between both modes for the same RouteHandle.

By default routesyncflag is set to ESMF_ROUTESYNC_BLOCKING in all of the RouteHandle execution methods, and the behavior is that of the VM-wide collective communication calls described in the previous sections. In the blocking mode the user must assume that the communication call will not return until all PETs have exchanged the precomputed information. On the other hand, the user has no guarantee about the exact synchronization behavior, and it is unsafe to make specific assumptions. What is guaranteed in the blocking communication mode is that when the call returns on the local PET, all data exchanges associated with all local DEs have finished. This means that all in-bound data elements are valid and that all out-bound data elements can safely be overwritten by the user.

```
call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=routehandle, routesyncflag=ESMF_ROUTESYNC_BLOCKING, rc=rc)
```

The same exchange pattern, that is encoded in routehandle, can be executed in non-blocking mode, simply by setting the appropriate routesyncflag when calling into ESMF_ArrayRedist().

At first sight there are obvious similarities between the non-blocking RouteHandle based execution paradigm and the non-blocking message passing calls provided by MPI. However, there are significant differences in the behavior of the non-blocking point-to-point calls that MPI defines and the non-blocking mode of the collective exchange patterns described by ESMF RouteHandles.

Setting routesyncflag to ESMF_ROUTESYNC_NBSTART in any RouteHandle execution call returns immediately after all out-bound data has been moved into ESMF internal transfer buffers and the exchange has been initiated.

```
call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=routehandle, routesyncflag=ESMF_ROUTESYNC_NBSTART, rc=rc)
```

Once a call with `routesyncflag = ESMF_ROUTESYNC_NBSTART` returns, it is safe to modify the out-bound data elements in the `srcArray` object. However, no guarantees are made for the in-bound data elements in `dstArray` at this phase of the non-blocking execution. It is unsafe to access these elements until the exchange has finished locally.

One way to ensure that the exchange has finished locally is to call with `routesyncflag` set to `ESMF_ROUTESYNC_NBWAITFINISH`.

```
call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=routehandle, routesyncflag=ESMF_ROUTESYNC_NBWAITFINISH, rc=rc)
```

Calling with `routesyncflag = ESMF_ROUTESYNC_NBWAITFINISH` instructs the communication method to wait and block until the previously started exchange has finished, and has been processed locally according to the RouteHandle. Once the call returns, it is safe to access both in-bound and out-bound data elements in `dstArray` and `srcArray`, respectively.

Some situations require more flexibility than is provided by the `ESMF_ROUTESYNC_NBSTART` - `ESMF_ROUTESYNC_NBWAITFINISH` pair. For instance, a Component that needs to interact with several other Components, virtually simultaneously, would initiated several different exchanges with `ESMF_ROUTESYNC_NBSTART`. Calling with `ESMF_ROUTESYNC_NBWAITFINISH` for any of the outstanding exchanges may potentially block for a long time, lowering the throughput. In the worst case a dead lock situation may arise. Calling with `routesyncflag = ESMF_ROUTESYNC_NBTESTFINISH` addresses this problem.

```
call ESMF_ArrayRedist(srcArray=srcArray, dstArray=dstArray, &
  routehandle=routehandle, routesyncflag=ESMF_ROUTESYNC_NBTESTFINISH, &
  finishedflag=finishflag, rc=rc)
```

This call tests the locally outstanding data transfer operation in `routehandle`, and finishes the exchange as much as currently possible. It does not block until the entire exchange has finished locally, instead it returns immediately after one round of testing has been completed. The optional return argument `finishedflag` is set to `.true.` if the exchange is completely finished locally, and set to `.false.` otherwise.

The user code must decide, depending on the value of the returned `finishedflag`, whether additional calls are required to finish an outstanding non-blocking exchange. If so, it can be done by calling `ESMF_ArrayRedist()` repeatedly with `ESMF_ROUTESYNC_NBTESTFINISH` until `finishedflag` comes back with a value of `.true.`. Such a loop allows other pieces of user code to be executed between the calls. A call with `ESMF_ROUTESYNC_NBWAITFINISH` can alternatively be used to block until the exchange has locally finished.

Noteworthy property. It is allowable to invoke a RouteHandle based communication call with `routesyncflag` set to `ESMF_ROUTESYNC_NBTESTFINISH` or `ESMF_ROUTESYNC_NBWAITFINISH` on a specific RouteHandle without there being an outstanding non-blocking exchange. As a matter of fact, it is not required that there was ever a call made with `ESMF_ROUTESYNC_NBSTART` for the RouteHandle. In these cases the calls made with `ESMF_ROUTESYNC_NBTESTFINISH` or `ESMF_ROUTESYNC_NBWAITFINISH` will simply return immediately (with `finishedflag` set to `.true.`).

Noteworthy property. It is fine to mix blocking and non-blocking invocations of the same RouteHandle based communication call across the PETs. This means that it is fine for some PETs to issue the call with `ESMF_ROUTESYNC_BLOCKING` (or using the default), while other PETs call the same communication call with `ESMF_ROUTESYNC_NBSTART`.

Noteworthy restriction. A RouteHandle that is currently involved in an outstanding non-blocking exchange may *not* be used to start any further exchanges, neither blocking nor non-blocking. This restriction is independent of whether the newly started RouteHandle based exchange is made for the same or for different data objects.

28.3 Restrictions and Future Work

- **CAUTION:** Depending on the specific `ESMF_ArrayCreate()` entry point used during Array creation, certain Fortran operations are not supported on the Fortran array pointer `farrayPtr`, returned by `ESMF_ArrayGet()`. Only if the `ESMF_ArrayCreate()` *from pointer* variant was used, will the returned `farrayPtr` variable contain the original bounds information, and be suitable for the Fortran `deallocate()` call. This limitation is a direct consequence of the Fortran 95 standard relating to the passing of array arguments. Fortran array pointers returned from an Array that was created through the *assumed shape array* variant of `ESMF_ArrayCreate()` will have bounds that are consistent with the other arguments specified during Array creation. These pointers are not suitable for deallocation in accordance to the Fortran 95 standard.
- **1D limit:** ArrayHalo(), ArrayRedist() and ArraySMM() operations on Arrays created on DistGrids with arbitrary sequence indices are currently limited to 1D arbitrary DistGrids. There is no restriction on the number, size and mapping of undistributed Array dimensions in the presence of such a 1D arbitrary DistGrid.

28.4 Design and Implementation Notes

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

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

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

28.5 Class API

28.5.1 ESMF_ArrayAssignment(=) - Array assignment

INTERFACE:

```
interface assignment(=)
array1 = array2
```

ARGUMENTS:

```
type(ESMF_Array) :: array1
type(ESMF_Array) :: array2
```

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:

Assign array1 as an alias to the same ESMF Array object in memory as array2. If array2 is invalid, then array1 will be equally invalid after the assignment.

The arguments are:

array1 The `ESMF_Array` object on the left hand side of the assignment.

array2 The `ESMF_Array` object on the right hand side of the assignment.

28.5.2 ESMF_ArrayOperator(==) - Array equality operator

INTERFACE:

```
interface operator(==)
if (array1 == array2) then ... endif
OR
result = (array1 == array2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array1
type(ESMF_Array), intent(in) :: array2
```

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:

Test whether array1 and array2 are valid aliases to the same ESMF Array object in memory. For a more general comparison of two ESMF Arrays, going beyond the simple alias test, the ESMF_ArrayMatch() function (not yet implemented) must be used.

The arguments are:

array1 The `ESMF_Array` object on the left hand side of the equality operation.

array2 The `ESMF_Array` object on the right hand side of the equality operation.

28.5.3 ESMF_ArrayOperator(/=) - Array not equal operator

INTERFACE:

```
interface operator(/=)
if (array1 /= array2) then ... endif
OR
result = (array1 /= array2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array1
type(ESMF_Array), intent(in) :: array2
```

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:

Test whether array1 and array2 are *not* valid aliases to the same ESMF Array object in memory. For a more general comparison of two ESMF Arrays, going beyond the simple alias test, the ESMF_ArrayMatch() function (not yet implemented) must be used.

The arguments are:

array1 The `ESMF_Array` object on the left hand side of the non-equality operation.

array2 The `ESMF_Array` object on the right hand side of the non-equality operation.

28.5.4 ESMF_ArrayCopy - Copy data from one Array object to another

INTERFACE:

```
subroutine ESMF_ArrayCopy(arrayOut, arrayIn, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Copy data from one ESMF_Array object to another.

The arguments are:

arrayOut ESMF_Array object into which to copy the data. The incoming arrayOut must already references a matching memory allocation.

arrayIn ESMF_Array object that holds the data to be copied.

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

28.5.5 ESMF_ArrayCreate - Create Array object from Fortran array pointer

INTERFACE:

```
! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateFrmPtr<rank><type><kind>(distgrid, farrayPtr, &
datacopyflag, distgridToArrayMap, computationalEdgeLWidth, &
computationalEdgeUWidth, computationalLWidth, &
computationalUWidth, totalLWidth, &
totalUWidth, name, rc)
```

RETURN VALUE:

```
type(ESMF_Array) :: ESMF_ArrayCreateDataPtr<rank><type><kind>
```

ARGUMENTS:

```
type(ESMF_DistGrid), intent(in) :: distgrid
<type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
integer, intent(in), optional :: distgridToArrayMap(:)
integer, intent(in), optional :: computationalEdgeLWidth(:)
integer, intent(in), optional :: computationalEdgeUWidth(:)
integer, intent(in), optional :: computationalLWidth(:)
integer, intent(in), optional :: computationalUWidth(:)
integer, intent(in), optional :: totalLWidth(:)
integer, intent(in), optional :: totalUWidth(:)
character (len=*), intent(in), optional :: name
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:

Create an ESMF_Array object from existing local native Fortran arrays with pointer attribute. The decomposition and distribution is specified by the distgrid argument. Each PET must issue this call with identical arguments in order to create a consistent Array object. The only exception is the farrayPtr argument which will be different on each PET. The bounds of the local arrays are preserved by this call and determine the bounds of the total region of the resulting Array object. Bounds of the DE-local exclusive regions are set to be consistent with the total regions and the specified distgrid argument. Bounds for Array dimensions that are not distributed are automatically set to the bounds provided by farrayPtr.

This interface requires a 1 DE per PET decomposition. The Array object will not be created and an error will be returned if this condition is not met.

The not distributed Array dimensions form a tensor of rank = array.rank - distgrid.dimCount. The widths of the computational region are set to the provided value, or zero by default, for all tensor elements. Use ESMF_ArraySet() to change these default settings after the Array object has been created.

The return value is the newly created ESMF_Array object.

The arguments are:

distgrid ESMF_DistGrid object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank of farrayPtr.

farrayPtr Valid native Fortran array with pointer attribute. Memory must be associated with the actual argument. The type/kind/rank information of farrayPtr will be used to set Array's properties accordingly. The shape of farrayPtr will be checked against the information contained in the distgrid. The bounds of farrayPtr will be preserved by this call and the bounds of the resulting Array object are set accordingly.

[datacopyflag] Specifies whether the Array object will reference the memory allocation provided by farrayPtr directly or will copy the data from farrayPtr into a new memory allocation. For valid values see **??**. The default is ESMF_DATACOPY_REFERENCE. Note that the ESMF_DATACOPY_REFERENCE option may not be safe when providing an array slice in farrayPtr.

[distgridToArrayMap] List that contains as many elements as is indicated by distgrids's dimCount. The list elements map each dimension of the DistGrid object to a dimension in farrayPtr by specifying the appropriate Array dimension index. The default is to map all of distgrid's dimensions against the lower dimensions of the farrayPtr argument in sequence, i.e. distgridToArrayMap = (/1, 2, .../). Unmapped farrayPtr dimensions are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All distgridToArrayMap entries must be greater than or equal to zero and smaller than or equal to the Array rank. It is erroneous to specify the same entry multiple times unless it is zero. If the Array rank is less than the DistGrid dimCount then the default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the distgridToArrayMap indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

[computationalEdgeLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region for DEs that are located on the edge of a tile. The default is a zero vector.

[computationalEdgeUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region for DEs that are located on the edge of a tile. The default is a zero vector.

[computationalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region. The default is a zero vector.

[computationalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region. The default is a zero vector.

[totalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the total memory region with respect to the lower corner of the exclusive region. The default is to accommodate the union of exclusive and computational region exactly.

[totalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the total memory region with respect to the upper corner of the exclusive region. The default is a vector that contains the remaining number of elements in each direction as to fit the union of exclusive and computational region into the memory region provided by the `farrayPtr` argument.

[name] Name of the Array object.

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

28.5.6 ESMF_ArrayCreate - Create Array object from Fortran array pointer w/ arbitrary seqIndices for halo

INTERFACE:

```
! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateFrmPtrArb<indexkind><rank><type><kind>(distgrid, &
farrayPtr, haloSeqIndexList, datacopyflag, &
distgridToArrayMap, name, rc)
```

RETURN VALUE:

```
type(ESMF_Array) :: ESMF_ArrayCreateDataPtrArb<rank><type><kind>
```

ARGUMENTS:

```
type(ESMF_DistGrid), intent(in) :: distgrid
<type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
integer(ESMF_KIND_<indexkind>), intent(in) :: haloSeqIndexList(:)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
integer, intent(in), optional :: distgridToArrayMap(:)
character (len=*), intent(in), optional :: name
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:

Create an ESMF_Array object from existing local native Fortran arrays with pointer attribute, according to distgrid. Besides farrayPtr each PET must issue this call with identical arguments in order to create a consistent Array object. The bounds of the local arrays are preserved by this call and determine the bounds of the total region of the resulting Array object. Bounds of the DE-local exclusive regions are set to be consistent with the total regions and the specified distgrid argument. Bounds for Array dimensions that are not distributed are automatically set to the bounds provided by farrayPtr.

This interface requires a 1 DE per PET decomposition. The Array object will not be created and an error will be returned if this condition is not met.

The not distributed Array dimensions form a tensor of rank = array.rank - distgrid.dimCount. The widths of the computational region are set to the provided value, or zero by default, for all tensor elements. Use ESMF_ArraySet() to change these default settings after the Array object has been created.

The return value is the newly created ESMF_Array object.

The arguments are:

distgrid ESMF_DistGrid object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank of farrayPtr.

farrayPtr Valid native Fortran array with pointer attribute. Memory must be associated with the actual argument. The type/kind/rank information of farrayPtr will be used to set Array's properties accordingly. The shape of farrayPtr will be checked against the information contained in the distgrid. The bounds of farrayPtr will be preserved by this call and the bounds of the resulting Array object are set accordingly.

haloSeqIndexList One dimensional array containing sequence indices of local halo region. The size (and content) of haloSeqIndexList can (and typically will) be different on each PET. The haloSeqIndexList argument is of integer type, but can be of different kind in order to support both 32-bit (ESMF_KIND_I4) and 64-bit (ESMF_KIND_I8) indexing.

[datacopyflag] Specifies whether the Array object will reference the memory allocation provided by farrayPtr directly or will copy the data from farrayPtr into a new memory allocation. For valid values see **??**. The default is ESMF_DATACOPY_REFERENCE. Note that the ESMF_DATACOPY_REFERENCE option may not be safe when providing an array slice in farrayPtr.

[distgridToArrayMap] List that contains as many elements as is indicated by distgrids's dimCount. The list elements map each dimension of the DistGrid object to a dimension in farrayPtr by specifying the appropriate Array dimension index. The default is to map all of distgrid's dimensions against the lower dimensions of the farrayPtr argument in sequence, i.e. distgridToArrayMap = (/1, 2, .../). Unmapped farrayPtr dimensions are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All distgridToArrayMap entries must be greater than or equal to zero and smaller than or equal to the Array rank. It is erroneous to specify the same entry multiple times unless it is zero. If the Array rank is less than the DistGrid dimCount then the default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the distgridToArrayMap indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

[name] Name of the Array object.

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

28.5.7 ESMF_ArrayCreate - Create Array object from Fortran array

INTERFACE:

```
! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateAsmdSp<rank><type><kind>(distgrid, farray, &
indexflag, datacopyflag, distgridToArrayMap, &
computationalEdgeLWidth, computationalEdgeUWidth, computationalLWidth, &
computationalUWidth, totalLWidth, &
totalUWidth, undistLBound, undistUBound, name, rc)
```

RETURN VALUE:

```
type(ESMF_Array) :: ESMF_ArrayCreateDataAssmdShape<rank><type><kind>
```

ARGUMENTS:

```
type(ESMF_DistGrid), intent(in) :: distgrid
<type> (ESMF_KIND_<kind>), intent(in), target :: farray(<rank>)
type(ESMF_Index_Flag), intent(in) :: indexflag
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
integer, intent(in), optional :: distgridToArrayMap(:)
integer, intent(in), optional :: computationalEdgeLWidth(:)
integer, intent(in), optional :: computationalEdgeUWidth(:)
integer, intent(in), optional :: computationalLWidth(:)
integer, intent(in), optional :: computationalUWidth(:)
integer, intent(in), optional :: totalLWidth(:)
integer, intent(in), optional :: totalUWidth(:)
integer, intent(in), optional :: undistLBound(:)
integer, intent(in), optional :: undistUBound(:)
character (len=*), intent(in), optional :: name
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:

Create an `ESMF_Array` object from an existing local native Fortran array. The decomposition and distribution is specified by the `distgrid` argument. Each PET must issue this call with identical arguments in order to create a consistent Array object. The only exception is the `farray` argument which will be different on each PET. The local arrays provided must be dimensioned according to the DE-local total region. Bounds of the exclusive regions are set as specified in the `distgrid` argument. Bounds for Array dimensions that are not distributed can be chosen freely using the `undistLBound` and `undistUBound` arguments.

This interface requires a 1 DE per PET decomposition. The Array object will not be created and an error will be returned if this condition is not met.

The not distributed Array dimensions form a tensor of rank = array.rank - distgrid.dimCount. The widths of the computational region are set to the provided value, or zero by default, for all tensor elements. Use ESMF_ArraySet() to change these default settings after the Array object has been created.

The return value is the newly created ESMF_Array object.

The arguments are:

distgrid ESMF_DistGrid object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank of farray.

farray Valid native Fortran array, i.e. memory must be associated with the actual argument. The type/kind/rank information of farray will be used to set Array's properties accordingly. The shape of farray will be checked against the information contained in the distgrid.

indexflag Indicate how DE-local indices are defined. See section **??** for a list of valid indexflag options.

[datacopyflag] Specifies whether the Array object will reference the memory allocation provided by farray directly or will copy the data from farray into a new memory allocation. For valid values see **??**. The default is ESMF_DATACOPY_REFERENCE. Note that the ESMF_DATACOPY_REFERENCE option may not be safe when providing an array slice in farray.

[distgridToArrayMap] List that contains as many elements as is indicated by distgrids's dimCount. The list elements map each dimension of the DistGrid object to a dimension in farray by specifying the appropriate Array dimension index. The default is to map all of distgrid's dimensions against the lower dimensions of the farray argument in sequence, i.e. distgridToArrayMap = (/1, 2, .../). Unmapped farray dimensions are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All distgridToArrayMap entries must be greater than or equal to zero and smaller than or equal to the Array rank. It is erroneous to specify the same entry multiple times unless it is zero. If the Array rank is less than the DistGrid dimCount then the default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the distgridToArrayMap indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

[computationalEdgeLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region for DEs that are located on the edge of a tile. The default is a zero vector.

[computationalEdgeUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region for DEs that are located on the edge of a tile. The default is a zero vector.

[computationalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region. The default is a zero vector.

[computationalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region. The default is a zero vector.

[totalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the total memory region with respect to the lower corner of the exclusive region. The default is to accommodate the union of exclusive and computational region exactly.

[totalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the total memory region with respect to the upper corner of the exclusive region. The default is a vector that contains the remaining number of elements in each direction as to fit the union of exclusive and computational region into the memory region provided by the farray argument.

[undistLBound] Lower bounds for the array dimensions that are not distributed. By default lbound is 1.

[undistUBound] Upper bounds for the array dimensions that are not distributed. By default ubound is equal to the extent of the corresponding dimension in `farray`.

[name] Name of the Array object.

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

28.5.8 ESMF_ArrayCreate - Create Array object from Fortran array w/ arbitrary seqIndices for halo

INTERFACE:

```
! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateAsmdSpArb<indexkind><rank><type><kind>(distgrid, &
farray, indexflag, haloSeqIndexList, datacopyflag, &
distgridToArrayMap, computationalEdgeLWidth, computationalEdgeUWidth, &
computationalLWidth, computationalUWidth, totalLWidth, totalUWidth, &
undistLBound, undistUBound, name, rc)
```

RETURN VALUE:

```
type(ESMF_Array) :: ESMF_ArrayCreateDataAssmdShapeArb<rank><type><kind>
```

ARGUMENTS:

```
type(ESMF_DistGrid), intent(in) :: distgrid
<type> (ESMF_KIND_<kind>), intent(in), target :: farray(<rank>)
type(ESMF_Index_Flag), intent(in) :: indexflag
integer(ESMF_KIND_<indexkind>), intent(in) :: haloSeqIndexList(:)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
integer, intent(in), optional :: distgridToArrayMap(:)
integer, intent(in), optional :: computationalEdgeLWidth(:)
integer, intent(in), optional :: computationalEdgeUWidth(:)
integer, intent(in), optional :: computationalLWidth(:)
integer, intent(in), optional :: computationalUWidth(:)
integer, intent(in), optional :: totalLWidth(:)
integer, intent(in), optional :: totalUWidth(:)
integer, intent(in), optional :: undistLBound(:)
integer, intent(in), optional :: undistUBound(:)
character (len=*), intent(in), optional :: name
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:

Create an ESMF_Array object from an existing local native Fortran array. The decomposition and distribution is specified by the distgrid argument. Each PET must issue this call with identical arguments in order to create a consistent Array object. The only exception is the farray argument which will be different on each PET. The local arrays provided must be dimensioned according to the DE-local total region. Bounds of the exclusive regions are set as specified in the distgrid argument. Bounds for Array dimensions that are not distributed can be chosen freely using the undistLBound and undistUBound arguments.

This interface requires a 1 DE per PET decomposition. The Array object will not be created and an error will be returned if this condition is not met.

The not distributed Array dimensions form a tensor of rank = array.rank - distgrid.dimCount. The widths of the computational region are set to the provided value, or zero by default, for all tensor elements. Use ESMF_ArraySet() to change these default settings after the Array object has been created.

The return value is the newly created ESMF_Array object.

The arguments are:

distgrid ESMF_DistGrid object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank of farray.

farray Valid native Fortran array, i.e. memory must be associated with the actual argument. The type/kind/rank information of farray will be used to set Array's properties accordingly. The shape of farray will be checked against the information contained in the distgrid.

indexflag Indicate how DE-local indices are defined. See section **??** for a list of valid indexflag options.

haloSeqIndexList One dimensional array containing sequence indices of local halo region. The size (and content) of haloSeqIndexList can (and typically will) be different on each PET. The haloSeqIndexList argument is of integer type, but can be of different kind in order to support both 32-bit (ESMF_KIND_I4) and 64-bit (ESMF_KIND_I8) indexing.

[datacopyflag] Specifies whether the Array object will reference the memory allocation provided by farray directly or will copy the data from farray into a new memory allocation. For valid values see **??**. The default is ESMF_DATACOPY_REFERENCE. Note that the ESMF_DATACOPY_REFERENCE option may not be safe when providing an array slice in farray.

[distgridToArrayMap] List that contains as many elements as is indicated by distgrids's dimCount. The list elements map each dimension of the DistGrid object to a dimension in farray by specifying the appropriate Array dimension index. The default is to map all of distgrid's dimensions against the lower dimensions of the farray argument in sequence, i.e. distgridToArrayMap = (/1, 2, .../). Unmapped farray dimensions are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All distgridToArrayMap entries must be greater than or equal to zero and smaller than or equal to the Array rank. It is erroneous to specify the same entry multiple times unless it is zero. If the Array rank is less than the DistGrid dimCount then the default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the distgridToArrayMap indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

[computationalEdgeLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region for DEs that are located on the edge of a tile. The default is a zero vector.

[computationalEdgeUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region for DEs that are located on the edge of a tile. The default is a zero vector.

[computationalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region. The default is a zero vector.

[computationalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region. The default is a zero vector.

[totalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the total memory region with respect to the lower corner of the exclusive region. The default is to accommodate the union of exclusive and computational region exactly.

[totalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the total memory region with respect to the upper corner of the exclusive region. The default is a vector that contains the remaining number of elements in each direction as to fit the union of exclusive and computational region into the memory region provided by the `farray` argument.

[undistLBound] Lower bounds for the array dimensions that are not distributed. By default lbound is 1.

[undistUBound] Upper bounds for the array dimensions that are not distributed. By default ubound is equal to the extent of the corresponding dimension in `farray`.

[name] Name of the Array object.

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

28.5.9 ESMF_ArrayCreate - Create Array object from a list of LocalArray objects

INTERFACE:

```
! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateLocalArray(distgrid, localarrayList, &
  indexflag, datacopyflag, distgridToArrayMap, computationalEdgeLWidth, &
  computationalEdgeUWidth, computationalLWidth, computationalUWidth, &
  totalLWidth, totalUWidth, undistLBound, undistUBound, name, rc)
```

RETURN VALUE:

```
type(ESMF_Array) :: ESMF_ArrayCreateLocalArray
```

ARGUMENTS:

```
      type(ESMF_DistGrid), intent(in) :: distgrid
      type(ESMF_LocalArray), intent(in) :: localarrayList(:)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_Index_Flag), intent(in), optional :: indexflag
      type(ESMF_DataCopy_Flag),intent(in), optional :: datacopyflag
      integer, intent(in), optional :: distgridToArrayMap(:)
      integer, intent(in), optional :: computationalEdgeLWidth(:)
      integer, intent(in), optional :: computationalEdgeUWidth(:)
      integer, intent(in), optional :: computationalLWidth(:)
      integer, intent(in), optional :: computationalUWidth(:)
```

```
integer, intent(in), optional :: totalLWidth(:)
integer, intent(in), optional :: totalUWidth(:)
integer, intent(in), optional :: undistLBound(:)
integer, intent(in), optional :: undistUBound(:)
character (len=*), intent(in), optional :: name
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:

Create an ESMF_Array object from existing ESMF_LocalArray objects. The decomposition and distribution is specified by the distgrid argument. Each PET must issue this call with identical arguments in order to create a consistent Array object. The only exception is the localarrayList argument which will be different on each PET. The local arrays provided must be dimensioned according to the DE-local total region. Bounds of the exclusive regions are set as specified in the distgrid argument. Bounds for Array dimensions that are not distributed can be chosen freely using the undistLBound and undistUBound arguments.

This interface is able to handle multiple DEs per PET.

The not distributed Array dimensions form a tensor of rank = array.rank - distgrid.dimCount. The widths of the computational region are set to the provided value, or zero by default, for all tensor elements. Use ESMF_ArraySet() to change these default settings after the Array object has been created.

The return value is the newly created ESMF_Array object.

The arguments are:

distgrid ESMF_DistGrid object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank specified in arrayspec, otherwise a runtime ESMF error will be raised.

localarrayList List of valid ESMF_LocalArray objects, i.e. memory must be associated with the actual arguments. The type/kind/rank information of all localarrayList elements must be identical and will be used to set Array's properties accordingly. The shape of each localarrayList element will be checked against the information contained in the distgrid.

[indexflag] Indicate how DE-local indices are defined. By default, the exclusive region of each DE is placed to start at the local index space origin, i.e. (1, 1, ..., 1). Alternatively the DE-local index space can be aligned with the global index space, if a global index space is well defined by the associated DistGrid. See section **??** for a list of valid indexflag options.

[datacopyflag] Specifies whether the Array object will reference the memory allocation of the arrays provided in localarrayList directly, or will copy the actual data into new memory allocations. For valid values see **??**. The default is ESMF_DATACOPY_REFERENCE.

[distgridToArrayMap] List that contains as many elements as is indicated by distgrids's dimCount. The list elements map each dimension of the DistGrid object to a dimension in the localarrayList elements by specifying the appropriate Array dimension index. The default is to map all of distgrid's dimensions against the lower dimensions of the localarrayList elements in sequence, i.e. distgridToArrayMap = (/1, 2, .../). Unmapped dimensions in the localarrayList elements are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All distgridToArrayMap entries must be greater than or equal to zero and smaller than or equal to the Array rank. It is erroneous to specify the same entry multiple times unless it is zero. If the Array rank is less than the DistGrid dimCount then the

default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the `distgridToArrayMap` indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

[computationalEdgeLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region for DEs that are located on the edge of a tile.

[computationalEdgeUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region for DEs that are located on the edge of a tile.

[computationalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region. The default is a zero vector.

[computationalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region. The default is a zero vector.

[totalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the total memory region with respect to the lower corner of the exclusive region. The default is to accommodate the union of exclusive and computational region exactly.

[totalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the total memory region with respect to the upper corner of the exclusive region. The default is a vector that contains the remaining number of elements in each direction as to fit the union of exclusive and computational region into the memory region provided by the `localarrayList` argument.

[undistLBound] Lower bounds for the array dimensions that are not distributed. By default lbound is 1.

[undistUBound] Upper bounds for the array dimensions that are not distributed. By default ubound is equal to the extent of the corresponding dimension in `localarrayList`.

[name] Name of the Array object.

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

28.5.10 ESMF_ArrayCreate - Create Array object from a list of LocalArray objects w/ arbitrary seqIndices for halo

INTERFACE:

```
! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateLocalArrayArb<indexkind>(distgrid, localarrayList, &
haloSeqIndexList, indexflag, datacopyflag, &
distgridToArrayMap, computationalEdgeLWidth, computationalEdgeUWidth, &
computationalLWidth, computationalUWidth, &
totalLWidth, totalUWidth, undistLBound, undistUBound, name, rc)
```

RETURN VALUE:

```
type(ESMF_Array) :: ESMF_ArrayCreateLocalArrayArb
```

ARGUMENTS:

```
   type(ESMF_DistGrid), intent(in) :: distgrid
   type(ESMF_LocalArray), intent(in) :: localarrayList(:)
   integer(ESMF_KIND_<indexkind>), intent(in) :: haloSeqIndexList(:)
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
   type(ESMF_Index_Flag), intent(in), optional :: indexflag
   type(ESMF_DataCopy_Flag),intent(in), optional :: datacopyflag
   integer, intent(in), optional :: distgridToArrayMap(:)
   integer, intent(in), optional :: computationalEdgeLWidth(:)
   integer, intent(in), optional :: computationalEdgeUWidth(:)
   integer, intent(in), optional :: computationalLWidth(:)
   integer, intent(in), optional :: computationalUWidth(:)
   integer, intent(in), optional :: totalLWidth(:)
   integer, intent(in), optional :: totalUWidth(:)
   integer, intent(in), optional :: undistLBound(:)
   integer, intent(in), optional :: undistUBound(:)
   character (len=*), intent(in), optional :: name
   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 arguments `indexflag`, `computationalEdgeLWidth`, `computationalEdgeUWidth`, `computationalLWidth`, `computationalUWidth`, `totalLWidth`, `totalUWidth`. These arguments were missed in previous versions by mistake.

DESCRIPTION:

Create an `ESMF_Array` object from existing `ESMF_LocalArray` objects according to distgrid. Each PET must issue this call in unison in order to create a consistent Array object. The local arrays provided must be dimensioned according to the DE-local total region. Bounds of the exclusive regions are set as specified in the distgrid argument. Bounds for array dimensions that are not distributed can be chosen freely using the `undistLBound` and `undistUBound` arguments.

The return value is the newly created `ESMF_Array` object.

The arguments are:

distgrid `ESMF_DistGrid` object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank specified in arrayspec, otherwise a runtime ESMF error will be raised.

localarrayList List of valid `ESMF_LocalArray` objects, i.e. memory must be associated with the actual arguments. The type/kind/rank information of all `localarrayList` elements must be identical and will be used to set `Array`'s properties accordingly. The shape of each `localarrayList` element will be checked against the information contained in the `distgrid`.

haloSeqIndexList One dimensional array containing sequence indices of local halo region. The size (and content) of `haloSeqIndexList` can (and typically will) be different on each PET. The `haloSeqIndexList` argument is of integer type, but can be of different kind in order to support both 32-bit (`ESMF_KIND_I4`) and 64-bit (`ESMF_KIND_I8`) indexing.

[indexflag] Indicate how DE-local indices are defined. By default, the exclusive region of each DE is placed to start at the local index space origin, i.e. (1, 1, ..., 1). Alternatively the DE-local index space can be aligned with the global index space, if a global index space is well defined by the associated DistGrid. See section **??** for a list of valid indexflag options.

[datacopyflag] Specifies whether the Array object will reference the memory allocation of the arrays provided in `localarrayList` directly, or will copy the actual data into new memory allocations. For valid values see **??**. The default is `ESMF_DATACOPY_REFERENCE`.

[distgridToArrayMap] List that contains as many elements as is indicated by `distgrids`'s dimCount. The list elements map each dimension of the DistGrid object to a dimension in the `localarrayList` elements by specifying the appropriate Array dimension index. The default is to map all of `distgrid`'s dimensions against the lower dimensions of the `localarrayList` elements in sequence, i.e. `distgridToArrayMap = (/1, 2, .../)`. Unmapped dimensions in the `localarrayList` elements are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All `distgridToArrayMap` entries must be greater than or equal to zero and smaller than or equal to the Array rank. It is erroneous to specify the same entry multiple times unless it is zero. If the Array rank is less than the DistGrid dimCount then the default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the `distgridToArrayMap` indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

[computationalEdgeLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region for DEs that are located on the edge of a tile.

[computationalEdgeUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region for DEs that are located on the edge of a tile.

[computationalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region. The default is a zero vector.

[computationalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region. The default is a zero vector.

[totalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the total memory region with respect to the lower corner of the exclusive region. The default is to accommodate the union of exclusive and computational region exactly.

[totalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the total memory region with respect to the upper corner of the exclusive region. The default is a vector that contains the remaining number of elements in each direction as to fit the union of exclusive and computational region into the memory region provided by the `localarrayList` argument.

[undistLBound] Lower bounds for the array dimensions that are not distributed. By default lbound is 1.

[undistUBound] Upper bounds for the array dimensions that are not distributed. By default ubound is equal to the extent of the corresponding dimension in `localarrayList`.

[name] Name of the Array object.

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

28.5.11 ESMF_ArrayCreate - Create Array object from typekind (allocate memory)

INTERFACE:

```
! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateAllocate(distgrid, typekind, &
  indexflag, pinflag, distgridToArrayMap, computationalEdgeLWidth, &
  computationalEdgeUWidth, computationalLWidth, computationalUWidth, &
  totalLWidth, totalUWidth, undistLBound, undistUBound, name, vm, rc)
```

RETURN VALUE:

```
type(ESMF_Array) :: ESMF_ArrayCreateAllocate
```

ARGUMENTS:

```
      type(ESMF_DistGrid), intent(in) :: distgrid
      type(ESMF_TypeKind_Flag), intent(in) :: typekind
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
      type(ESMF_Index_Flag), intent(in), optional :: indexflag
      type(ESMF_Pin_Flag), intent(in), optional :: pinflag
      integer, intent(in), optional :: distgridToArrayMap(:)
      integer, intent(in), optional :: computationalEdgeLWidth(:)
      integer, intent(in), optional :: computationalEdgeUWidth(:)
      integer, intent(in), optional :: computationalLWidth(:)
      integer, intent(in), optional :: computationalUWidth(:)
      integer, intent(in), optional :: totalLWidth(:)
      integer, intent(in), optional :: totalUWidth(:)
      integer, intent(in), optional :: undistLBound(:)
      integer, intent(in), optional :: undistUBound(:)
      character (len=*), intent(in), optional :: name
      type(ESMF_VM), intent(in), 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:

  6.3.0r Added argument `vm` to support object creation on a different VM than that of the current context.

  8.0.0 Added argument `pinflag` to provide access to DE sharing between PETs.

DESCRIPTION:

Create an `ESMF_Array` object and allocate uninitialized data space according to typekind and distgrid. The Array rank is indirectly determined by the incoming information. Each PET must issue this call in unison in order to create

a consistent Array object. DE-local allocations are made according to the total region defined by the `distgrid` and the optional `Width` arguments.

The return value is the newly created `ESMF_Array` object.

The arguments are:

distgrid `ESMF_DistGrid` object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank specified in arrayspec, otherwise a runtime ESMF error will be raised.

typekind The typekind of the Array. See section **??** for a list of valid typekind options.

[indexflag] Indicate how DE-local indices are defined. By default, the exclusive region of each DE is placed to start at the local index space origin, i.e. (1, 1, ..., 1). Alternatively the DE-local index space can be aligned with the global index space, if a global index space is well defined by the associated DistGrid. See section **??** for a list of valid indexflag options.

[pinflag] Specify which type of resource DEs are pinned to. See section **??** for a list of valid pinning options. The default is to pin DEs to PETs, i.e. only the PET on which a DE was created considers the DE as local.

[distgridToArrayMap] List that contains as many elements as is indicated by `distgrids`'s dimCount. The list elements map each dimension of the DistGrid object to a dimension in the newly allocated Array object by specifying the appropriate Array dimension index. The default is to map all of `distgrid`'s dimensions against the lower dimensions of the Array object in sequence, i.e. `distgridToArrayMap = (/1, 2, .../)`. Unmapped dimensions in the Array object are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All `distgridToArrayMap` entries must be greater than or equal to zero and smaller than or equal to the Array rank. It is erroneous to specify the same entry multiple times unless it is zero. If the Array rank is less than the DistGrid dimCount then the default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the `distgridToArrayMap` indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

[computationalEdgeLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region for DEs that are located on the edge of a tile.

[computationalEdgeUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region for DEs that are located on the edge of a tile.

[computationalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region. The default is a zero vector.

[computationalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region. The default is a zero vector.

[totalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the total memory region with respect to the lower corner of the exclusive region. The default is to accommodate the union of exclusive and computational region.

[totalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the total memory region with respect to the upper corner of the exclusive region. The default is to accommodate the union of exclusive and computational region.

[undistLBound] Lower bounds for the array dimensions that are not distributed.

[undistUBound] Upper bounds for the array dimensions that are not distributed.

[name] Name of the Array object.

[vm] If present, the Array object is created on the specified ESMF_VM object. The default is to create on the VM of the current context.

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

28.5.12 ESMF_ArrayCreate - Create Array object from typekind (allocate memory) w/ arbitrary seqIndices for halo

INTERFACE:

```
! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateAllocateArb<indexkind>(distgrid, typekind, &
haloSeqIndexList, pinflag, distgridToArrayMap, &
undistLBound, undistUBound, name, rc)
```

RETURN VALUE:

```
type(ESMF_Array) :: ESMF_ArrayCreateAllocateArb
```

ARGUMENTS:

```
type(ESMF_DistGrid), intent(in) :: distgrid
type(ESMF_TypeKind_Flag), intent(in) :: typekind
integer(ESMF_KIND_<indexkind>), intent(in) :: haloSeqIndexList(:)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_Pin_Flag), intent(in), optional :: pinflag
integer, intent(in), optional :: distgridToArrayMap(:)
integer, intent(in), optional :: undistLBound(:)
integer, intent(in), optional :: undistUBound(:)
character (len=*), intent(in), optional :: name
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:

  8.0.0 Added argument pinflag to provide access to DE sharing between PETs.

DESCRIPTION:

Create an ESMF_Array object and allocate uninitialized data space according to typekind and distgrid. The Array rank is indirectly determined by the incoming information. Each PET must issue this call in unison in order to create a consistent Array object. DE-local allocations are made according to the total region defined by the distgrid and haloSeqIndexList arguments.

The return value is the newly created ESMF_Array object.

The arguments are:

distgrid ESMF_DistGrid object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank specified in arrayspec, otherwise a runtime ESMF error will be raised.

typekind The typekind of the Array. See section **??** for a list of valid typekind options.

haloSeqIndexList One dimensional array containing sequence indices of local halo region. The size (and content) of haloSeqIndexList can (and typically will) be different on each PET. The haloSeqIndexList argument is of integer type, but can be of different kind in order to support both 32-bit (ESMF_KIND_I4) and 64-bit (ESMF_KIND_I8) indexing.

[pinflag] Specify which type of resource DEs are pinned to. See section **??** for a list of valid pinning options. The default is to pin DEs to PETs, i.e. only the PET on which a DE was created considers the DE as local.

[distgridToArrayMap] List that contains as many elements as is indicated by distgrids's dimCount. The list elements map each dimension of the DistGrid object to a dimension in the newly allocated Array object by specifying the appropriate Array dimension index. The default is to map all of distgrid's dimensions against the lower dimensions of the Array object in sequence, i.e. distgridToArrayMap = (/1, 2, .../). Unmapped dimensions in the Array object are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All distgridToArrayMap entries must be greater than or equal to zero and smaller than or equal to the Array rank. It is erroneous to specify the same entry multiple times unless it is zero. If the Array rank is less than the DistGrid dimCount then the default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the distgridToArrayMap indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

[undistLBound] Lower bounds for the array dimensions that are not distributed.

[undistUBound] Upper bounds for the array dimensions that are not distributed.

[name] Name of the Array object.

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

28.5.13 ESMF_ArrayCreate - Create Array object from ArraySpec (allocate memory)

INTERFACE:

```
! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateAllocateAS(distgrid, arrayspec, &
  indexflag, pinflag, distgridToArrayMap, computationalEdgeLWidth, &
  computationalEdgeUWidth, computationalLWidth, computationalUWidth, &
  totalLWidth, totalUWidth, undistLBound, undistUBound, name, vm, rc)
```

RETURN VALUE:

```
type(ESMF_Array) :: ESMF_ArrayCreateAllocateAS
```

ARGUMENTS:

```
        type(ESMF_DistGrid), intent(in) :: distgrid
        type(ESMF_ArraySpec), intent(in) :: arrayspec
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
        type(ESMF_Index_Flag), intent(in), optional :: indexflag
        type(ESMF_Pin_Flag), intent(in), optional :: pinflag
        integer, intent(in), optional :: distgridToArrayMap(:)
        integer, intent(in), optional :: computationalEdgeLWidth(:)
        integer, intent(in), optional :: computationalEdgeUWidth(:)
        integer, intent(in), optional :: computationalLWidth(:)
        integer, intent(in), optional :: computationalUWidth(:)
        integer, intent(in), optional :: totalLWidth(:)
        integer, intent(in), optional :: totalUWidth(:)
        integer, intent(in), optional :: undistLBound(:)
        integer, intent(in), optional :: undistUBound(:)
        character (len=*), intent(in), optional :: name
        type(ESMF_VM), intent(in), 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:

  6.3.0r Added argument `vm` to support object creation on a different VM than that of the current context.

  8.0.0 Added argument `pinflag` to provide access to DE sharing between PETs.

DESCRIPTION:

Create an `ESMF_Array` object and allocate uninitialized data space according to arrayspec and distgrid. Each PET must issue this call with identical arguments in order to create a consistent Array object. DE-local allocations are made according to the total region defined by the arguments to this call: `distgrid` and the optional `Width` arguments.

The return value is the newly created `ESMF_Array` object.

The arguments are:

distgrid `ESMF_DistGrid` object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank specified in arrayspec, otherwise a runtime ESMF error will be raised.

arrayspec `ESMF_ArraySpec` object containing the type/kind/rank information.

[indexflag] Indicate how DE-local indices are defined. By default, the exclusive region of each DE is placed to start at the local index space origin, i.e. (1, 1, ..., 1). Alternatively the DE-local index space can be aligned with the global index space, if a global index space is well defined by the associated DistGrid. See section **??** for a list of valid indexflag options.

[pinflag] Specify which type of resource DEs are pinned to. See section **??** for a list of valid pinning options. The default is to pin DEs to PETs, i.e. only the PET on which a DE was created considers the DE as local.

[distgridToArrayMap] List that contains as many elements as is indicated by `distgrids`'s dimCount. The list elements map each dimension of the DistGrid object to a dimension in the newly allocated Array object by specifying the appropriate Array dimension index. The default is to map all of `distgrid`'s dimensions against the lower dimensions of the Array object in sequence, i.e. `distgridToArrayMap = (/1, 2, .../)`. Unmapped dimensions in the Array object are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All `distgridToArrayMap` entries must be greater than or equal to zero and smaller than or equal to the Array rank. It is erroneous to specify the same entry multiple times unless it is zero. If the Array rank is less than the DistGrid dimCount then the default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the `distgridToArrayMap` indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

[computationalEdgeLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region for DEs that are located on the edge of a tile.

[computationalEdgeUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region for DEs that are located on the edge of a tile.

[computationalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the computational region with respect to the lower corner of the exclusive region. The default is a zero vector.

[computationalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the computational region with respect to the upper corner of the exclusive region. The default is a zero vector.

[totalLWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the lower corner of the total memory region with respect to the lower corner of the exclusive region. The default is to accommodate the union of exclusive and computational region.

[totalUWidth] This vector argument must have dimCount elements, where dimCount is specified in distgrid. It specifies the upper corner of the total memory region with respect to the upper corner of the exclusive region. The default is to accommodate the union of exclusive and computational region.

[undistLBound] Lower bounds for the array dimensions that are not distributed.

[undistUBound] Upper bounds for the array dimensions that are not distributed.

[name] Name of the Array object.

[vm] If present, the Array object is created on the specified `ESMF_VM` object. The default is to create on the VM of the current context.

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

28.5.14 ESMF_ArrayCreate - Create Array object from ArraySpec (allocate memory) w/ arbitrary seqIndices for halo

INTERFACE:

```
! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateAllocateASArb<indexkind>(distgrid, arrayspec, &
haloSeqIndexList, pinflag, distgridToArrayMap, &
undistLBound, undistUBound, name, rc)
```

RETURN VALUE:

```
type(ESMF_Array) :: ESMF_ArrayCreateAllocateASArb
```

ARGUMENTS:

```
type(ESMF_DistGrid), intent(in) :: distgrid
type(ESMF_ArraySpec), intent(in) :: arrayspec
integer(ESMF_KIND_<indexkind>), intent(in) :: haloSeqIndexList(:)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_Pin_Flag), intent(in), optional :: pinflag
integer, intent(in), optional :: distgridToArrayMap(:)
integer, intent(in), optional :: undistLBound(:)
integer, intent(in), optional :: undistUBound(:)
character (len=*), intent(in), optional :: name
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:

  8.0.0 Added argument `pinflag` to provide access to DE sharing between PETs.

DESCRIPTION:

Create an `ESMF_Array` object and allocate uninitialized data space according to arrayspec and distgrid. Each PET must issue this call in unison in order to create a consistent Array object. DE-local allocations are made according to the total region defined by the arguments to this call: `distgrid` and `haloSeqIndexList` arguments.

The return value is the newly created `ESMF_Array` object.

The arguments are:

distgrid `ESMF_DistGrid` object that describes how the array is decomposed and distributed over DEs. The dimCount of distgrid must be smaller or equal to the rank specified in arrayspec, otherwise a runtime ESMF error will be raised.

arrayspec `ESMF_ArraySpec` object containing the type/kind/rank information.

haloSeqIndexList One dimensional array containing sequence indices of local halo region. The size (and content) of `haloSeqIndexList` can (and typically will) be different on each PET. The `haloSeqIndexList` argument is of integer type, but can be of different kind in order to support both 32-bit (`ESMF_KIND_I4`) and 64-bit (`ESMF_KIND_I8`) indexing.

[pinflag] Specify which type of resource DEs are pinned to. See section **??** for a list of valid pinning options. The default is to pin DEs to PETs, i.e. only the PET on which a DE was created considers the DE as local.

[distgridToArrayMap] List that contains as many elements as is indicated by `distgrids`'s dimCount. The list elements map each dimension of the DistGrid object to a dimension in the newly allocated Array object by specifying the appropriate Array dimension index. The default is to map all of `distgrid`'s dimensions against the lower dimensions of the Array object in sequence, i.e. `distgridToArrayMap = (/1, 2, .../)`. Unmapped dimensions in the Array object are not decomposed dimensions and form a tensor of rank = Array.rank - DistGrid.dimCount. All `distgridToArrayMap` entries must be greater than or equal to zero and smaller than or equal to the Array rank. It is erroneous to specify the same entry multiple times unless it is zero. If the Array rank is less than the DistGrid dimCount then the default distgridToArrayMap will contain zeros for the dimCount - rank rightmost entries. A zero entry in the `distgridToArrayMap` indicates that the particular DistGrid dimension will be replicating the Array across the DEs along this direction.

[undistLBound] Lower bounds for the array dimensions that are not distributed.

[undistUBound] Upper bounds for the array dimensions that are not distributed.

[name] Name of the Array object.

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

28.5.15 ESMF_ArrayCreate - Create Array object as copy of existing Array object

INTERFACE:

```
! Private name; call using ESMF_ArrayCreate()
function ESMF_ArrayCreateCopy(array, datacopyflag, delayout, rc)
```

RETURN VALUE:

```
type(ESMF_Array) :: ESMF_ArrayCreateCopy
```

ARGUMENTS:

```
        type(ESMF_Array), intent(in) :: array
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
        type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
        type(ESMF_DELayout), intent(in), optional :: delayout
        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:

8.1.0 Added argument `datacopyflag` to select between different copy options.
Added argument `delayout` to create Array with different localDe -> DE mapping. This is identical to a change in DE -> PET mapping.

DESCRIPTION:

Create an `ESMF_Array` object as the copy of an existing Array.

The return value is the newly created `ESMF_Array` object.

The arguments are:

array `ESMF_Array` object to be copied.

[datacopyflag] Specifies whether the created Array object references the memory allocation provided by `array` directly or copies the data from `array` into a new memory allocation. For valid values see **??**. The default is `ESMF_DATACOPY_VALUE`.

[delayout] If present, override the DELayout of the incoming `distgrid`. By default use the DELayout defined in `distgrid`.

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

28.5.16 ESMF_ArrayDestroy - Release resources associated with an Array object

INTERFACE:

```
subroutine ESMF_ArrayDestroy(array, noGarbage, rc)
```

ARGUMENTS:

```
 type(ESMF_Array), intent(inout) :: array
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
 logical, intent(in), optional :: noGarbage
 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 `noGarbage`. The argument provides a mechanism to override the default garbage collection mechanism when destroying an ESMF object.

DESCRIPTION:

Destroy an ESMF_Array, releasing the resources associated with the object.

By default a small remnant of the object is kept in memory in order to prevent problems with dangling aliases. The default garbage collection mechanism can be overridden with the noGarbage argument.

The arguments are:

array ESMF_Array object to be destroyed.

[noGarbage] If set to .TRUE. the object will be fully destroyed and removed from the ESMF garbage collection system. Note however that under this condition ESMF cannot protect against accessing the destroyed object through dangling aliases – a situation which may lead to hard to debug application crashes.

It is generally recommended to leave the noGarbage argument set to .FALSE. (the default), and to take advantage of the ESMF garbage collection system which will prevent problems with dangling aliases or incorrect sequences of destroy calls. However this level of support requires that a small remnant of the object is kept in memory past the destroy call. This can lead to an unexpected increase in memory consumption over the course of execution in applications that use temporary ESMF objects. For situations where the repeated creation and destruction of temporary objects leads to memory issues, it is recommended to call with noGarbage set to .TRUE., fully removing the entire temporary object from memory.

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

28.5.17 ESMF_ArrayGather - Gather a Fortran array from an ESMF_Array

INTERFACE:

```
subroutine ESMF_ArrayGather(array, farray, rootPet, tile, vm, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(in) :: array
<type>(ESMF_KIND_<kind>), intent(out), target :: farray(<rank>)
integer, intent(in) :: rootPet
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
integer, intent(in), optional :: tile
type(ESMF_VM), intent(in), 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.

DESCRIPTION:

Gather the data of an ESMF_Array object into the `farray` located on `rootPET`. A single DistGrid tile of `array` must be gathered into `farray`. The optional `tile` argument allows selection of the tile. For Arrays defined on a single tile DistGrid the default selection (tile 1) will be correct. The shape of `farray` must match the shape of the tile in Array.

If the Array contains replicating DistGrid dimensions data will be gathered from the numerically higher DEs. Replicated data elements in numerically lower DEs will be ignored.

This version of the interface implements the PET-based blocking paradigm: Each PET of the VM must issue this call exactly once for *all* of its DEs. The call will block until all PET-local data objects are accessible.

The arguments are:

array The `ESMF_Array` object from which data will be gathered.

{farray} The Fortran array into which to gather data. Only root must provide a valid `farray`, the other PETs may treat `farray` as an optional argument.

rootPet PET that holds the valid destination array, i.e. `farray`.

[tile] The DistGrid tile in `array` from which to gather `farray`. By default `farray` will be gathered from tile 1.

[vm] Optional `ESMF_VM` object of the current context. Providing the VM of the current context will lower the method's overhead.

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

28.5.18 ESMF_ArrayGet - Get object-wide Array information

INTERFACE:

```
! Private name; call using ESMF_ArrayGet()
subroutine ESMF_ArrayGetDefault(array, arrayspec, typekind, &
  rank, localarrayList, indexflag, distgridToArrayMap, &
  distgridToPackedArrayMap, arrayToDistGridMap, undistLBound, &
  undistUBound, exclusiveLBound, exclusiveUBound, computationalLBound, &
  computationalUBound, totalLBound, totalUBound, computationalLWidth, &
  computationalUWidth, totalLWidth, totalUWidth, distgrid, dimCount, &
  tileCount, minIndexPTile, maxIndexPTile, deToTileMap, indexCountPDe, &
  delayout, deCount, localDeCount, ssiLocalDeCount, localDeToDeMap, &
  localDeList, & ! DEPRECATED ARGUMENT
  name, vm, rc)
```

ARGUMENTS:

```
  type(ESMF_Array), intent(in) :: array
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  type(ESMF_ArraySpec), intent(out), optional :: arrayspec
  type(ESMF_TypeKind_Flag), intent(out), optional :: typekind
  integer, intent(out), optional :: rank
  type(ESMF_LocalArray), target, intent(out), optional :: localarrayList(:)
  type(ESMF_Index_Flag), intent(out), optional :: indexflag
```

```
integer, target, intent(out), optional :: distgridToArrayMap(:)
integer, target, intent(out), optional :: distgridToPackedArrayMap(:)
integer, target, intent(out), optional :: arrayToDistGridMap(:)
integer, target, intent(out), optional :: undistLBound(:)
integer, target, intent(out), optional :: undistUBound(:)
integer, target, intent(out), optional :: exclusiveLBound(:,:)
integer, target, intent(out), optional :: exclusiveUBound(:,:)
integer, target, intent(out), optional :: computationalLBound(:,:)
integer, target, intent(out), optional :: computationalUBound(:,:)
integer, target, intent(out), optional :: totalLBound(:,:)
integer, target, intent(out), optional :: totalUBound(:,:)
integer, target, intent(out), optional :: computationalLWidth(:,:)
integer, target, intent(out), optional :: computationalUWidth(:,:)
integer, target, intent(out), optional :: totalLWidth(:,:)
integer, target, intent(out), optional :: totalUWidth(:,:)
type(ESMF_DistGrid), intent(out), optional :: distgrid
integer, intent(out), optional :: dimCount
integer, intent(out), optional :: tileCount
integer, intent(out), optional :: minIndexPTile(:,:)
integer, intent(out), optional :: maxIndexPTile(:,:)
integer, intent(out), optional :: deToTileMap(:)
integer, intent(out), optional :: indexCountPDe(:,:)
type(ESMF_DELayout), intent(out), optional :: delayout
integer, intent(out), optional :: deCount
integer, intent(out), optional :: localDeCount
integer, intent(out), optional :: ssiLocalDeCount
integer, intent(out), optional :: localDeToDeMap(:)
integer, intent(out), optional :: localDeList(:) ! DEPRECATED ARGUMENT
character(len=*), intent(out), optional :: name
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:

  5.2.0rp1 Added argument `localDeToDeMap`. Started to deprecate argument `localDeList`. The new argument name correctly uses the `Map` suffix and better describes the returned information. This was pointed out by user request.

  8.0.0 Added argument `ssiLocalDeCount` to support DE sharing between PETs on the same single system image (SSI).
  Added argument `vm` in order to offer information about the VM on which the Array was created.

DESCRIPTION:

Get internal information.

This interface works for any number of DEs per PET.

The arguments are:

array Queried `ESMF_Array` object.

[arrayspec] `ESMF_ArraySpec` object containing the type/kind/rank information of the Array object.

[typekind] TypeKind of the Array object.

[rank] Rank of the Array object.

[localarrayList] Upon return this holds a list of the associated `ESMC_LocalArray` objects. `localarrayList` must be allocated to be of size `localDeCount` or `ssiLocalDeCount`.

[indexflag] Upon return this flag indicates how the DE-local indices are defined. See section **??** for a list of possible return values.

[distgridToArrayMap] Upon return this list holds the Array dimensions against which the DistGrid dimensions are mapped. `distgridToArrayMap` must be allocated to be of size `dimCount`. An entry of zero indicates that the respective DistGrid dimension is replicating the Array across the DEs along this direction.

[distgridToPackedArrayMap] Upon return this list holds the indices of the Array dimensions in packed format against which the DistGrid dimensions are mapped. `distgridToPackedArrayMap` must be allocated to be of size `dimCount`. An entry of zero indicates that the respective DistGrid dimension is replicating the Array across the DEs along this direction.

[arrayToDistGridMap] Upon return this list holds the DistGrid dimensions against which the Array dimensions are mapped. `arrayToDistGridMap` must be allocated to be of size `rank`. An entry of zero indicates that the respective Array dimension is not decomposed, rendering it a tensor dimension.

[undistLBound] Upon return this array holds the lower bounds of the undistributed dimensions of the Array. `UndistLBound` must be allocated to be of size `rank-dimCount`.

[undistUBound] Upon return this array holds the upper bounds of the undistributed dimensions of the Array. `UndistUBound` must be allocated to be of size `rank-dimCount`.

[exclusiveLBound] Upon return this holds the lower bounds of the exclusive regions for all PET-local DEs. `exclusiveLBound` must be allocated to be of size `(dimCount, localDeCount)` or `(dimCount, ssiLocalDeCount)`.

[exclusiveUBound] Upon return this holds the upper bounds of the exclusive regions for all PET-local DEs. `exclusiveUBound` must be allocated to be of size `(dimCount, localDeCount)` or `(dimCount, ssiLocalDeCount)`.

[computationalLBound] Upon return this holds the lower bounds of the computational regions for all PET-local DEs. `computationalLBound` must be allocated to be of size `(dimCount, localDeCount)` or `(dimCount, ssiLocalDeCount)`.

[computationalUBound] Upon return this holds the upper bounds of the computational regions for all PET-local DEs. `computationalUBound` must be allocated to be of size `(dimCount, localDeCount)` or `(dimCount, ssiLocalDeCount)`.

[totalLBound] Upon return this holds the lower bounds of the total regions for all PET-local DEs. `totalLBound` must be allocated to be of size `(dimCount, localDeCount)` or `(dimCount, ssiLocalDeCount)`.

[totalUBound] Upon return this holds the upper bounds of the total regions for all PET-local DEs. `totalUBound` must be allocated to be of size `(dimCount, localDeCount)` or `(dimCount, ssiLocalDeCount)`.

[computationalLWidth] Upon return this holds the lower width of the computational regions for all PET-local DEs. `computationalLWidth` must be allocated to be of size `(dimCount, localDeCount)` or `(dimCount, ssiLocalDeCount)`.

[computationalUWidth] Upon return this holds the upper width of the computational regions for all PET-local DEs. `computationalUWidth` must be allocated to be of size `(dimCount, localDeCount)` or `(dimCount, ssiLocalDeCount)`.

[totalLWidth] Upon return this holds the lower width of the total memory regions for all PET-local DEs. `totalLWidth` must be allocated to be of size `(dimCount, localDeCount)` or `(dimCount, ssiLocalDeCount)`.

[totalUWidth] Upon return this holds the upper width of the total memory regions for all PET-local DEs. `totalUWidth` must be allocated to be of size `(dimCount, localDeCount)` or `(dimCount, ssiLocalDeCount)`.

[distgrid] Upon return this holds the associated `ESMF_DistGrid` object.

[dimCount] Number of dimensions (rank) of `distgrid`.

[tileCount] Number of tiles in `distgrid`.

[minIndexPTile] Lower index space corner per `dim`, per `tile`, with `size(minIndexPTile) == (/dimCount, tileCount/)`.

[maxIndexPTile] Upper index space corner per `dim`, per `tile`, with `size(maxIndexPTile) == (/dimCount, tileCount/)`.

[deToTileMap] List of tile id numbers, one for each DE, with `size(deToTileMap) == (/deCount/)`

[indexCountPDe] Array of extents per `dim`, per `de`, with `size(indexCountPDe) == (/dimCount, deCount/)`.

[delayout] The associated `ESMF_DELayout` object.

[deCount] The total number of DEs in the Array.

[localDeCount] The number of DEs in the Array associated with the local PET.

[ssiLocalDeCount] The number of DEs in the Array available to the local PET. This includes DEs that are local to other PETs on the same SSI, that are accessible via shared memory.

[localDeToDeMap] Mapping between localDe indices and the (global) DEs associated with the local PET. The localDe index variables are discussed in sections **??** and 28.2.5. The provided actual argument must be of size `localDeCount`, or `ssiLocalDeCount`, and will be filled accordingly.

[localDeList] **DEPRECATED ARGUMENT!** Please use the argument `localDeToDeMap` instead.

[name] Name of the Array object.

[vm The VM on which the Array object was created.

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

28.5.19 ESMF_ArrayGet - Get DE-local Array information for a specific dimension

INTERFACE:

```
! Private name; call using ESMF_ArrayGet()
subroutine ESMF_ArrayGetPLocalDePDim(array, dim, localDe, &
  indexCount, indexList, rc)
```

ARGUMENTS:

```
     type(ESMF_Array), intent(in) :: array
     integer, intent(in) :: dim
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer, intent(in), optional :: localDe
     integer, intent(out), optional :: indexCount
     integer, intent(out), optional :: indexList(:)
     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:

Get internal information per local DE, per dim.

This interface works for any number of DEs per PET.

The arguments are:

array Queried `ESMF_Array` object.

dim Dimension for which information is requested. `[1,..,dimCount]`

[localDe] Local DE for which information is requested. `[0,..,localDeCount-1]`. For `localDeCount==1` the `localDe` argument may be omitted, in which case it will default to `localDe=0`.

[indexCount] DistGrid indexCount associated with `localDe, dim`.

[indexList] List of DistGrid tile-local indices for `localDe` along dimension `dim`.

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

28.5.20 ESMF_ArrayGet - Get a DE-local Fortran array pointer from an Array

INTERFACE:

```
   ! Private name; call using ESMF_ArrayGet()
   subroutine ESMF_ArrayGetFPtr<rank><type><kind>(array, localDe, &
   farrayPtr, rc)
```

ARGUMENTS:

```
   type(ESMF_Array), intent(in) :: array
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
   integer, intent(in), optional :: localDe
   <type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
   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:

Access Fortran array pointer to the specified DE-local memory allocation of the Array object.

The arguments are:

array Queried `ESMF_Array` object.

[localDe] Local DE for which information is requested. `[0,..,localDeCount-1]`. For `localDeCount==1` the `localDe` argument may be omitted, in which case it will default to `localDe=0`.

farrayPtr Upon return, `farrayPtr` points to the DE-local data allocation of `localDe` in `array`. It depends on the specific entry point of `ESMF_ArrayCreate()` used during `array` creation, which Fortran operations are supported on the returned `farrayPtr`. See 28.3 for more details.

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

28.5.21 ESMF_ArrayGet - Get a DE-local LocalArray object from an Array

INTERFACE:

```
! Private name; call using ESMF_ArrayGet()
subroutine ESMF_ArrayGetLocalArray(array, localDe, localarray, rc)
```

ARGUMENTS:

```
    type(ESMF_Array), intent(in) :: array
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    integer, intent(in), optional :: localDe
    type(ESMF_LocalArray), intent(inout) :: localarray
    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:

Provide access to `ESMF_LocalArray` object that holds data for the specified local DE.

The arguments are:

array Queried `ESMF_Array` object.

[localDe] Local DE for which information is requested. `[0,..,localDeCount-1]`. For `localDeCount==1` the `localDe` argument may be omitted, in which case it will default to `localDe=0`.

localarray Upon return `localarray` refers to the DE-local data allocation of `array`.

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

28.5.22 ESMF_ArrayHalo - Execute an Array halo operation

INTERFACE:

```
   subroutine ESMF_ArrayHalo(array, routehandle, &
     routesyncflag, finishedflag, cancelledflag, checkflag, rc)
```

ARGUMENTS:

```
     type(ESMF_Array),          intent(inout)         :: array
     type(ESMF_RouteHandle),    intent(inout)         :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_RouteSync_Flag), intent(in),  optional :: routesyncflag
     logical,                   intent(out), optional :: finishedflag
     logical,                   intent(out), optional :: cancelledflag
     logical,                   intent(in),  optional :: checkflag
     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:

Execute a precomputed Array halo operation for `array`. The `array` argument must match the respective Array used during `ESMF_ArrayHaloStore()` in *type*, *kind*, and memory layout of the *distributed* dimensions. However, the size, number, and index order of *undistributed* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

See `ESMF_ArrayHaloStore()` on how to precompute `routehandle`.

This call is *collective* across the current VM.

array `ESMF_Array` containing data to be haloed.

routehandle Handle to the precomputed Route.

[routesyncflag] Indicate communication option. Default is `ESMF_ROUTESYNC_BLOCKING`, resulting in a blocking operation. See section **??** for a complete list of valid settings.

[finishedflag] Used in combination with `routesyncflag = ESMF_ROUTESYNC_NBTESTFINISH`. Returned `finishedflag` equal to `.true.` indicates that all operations have finished. A value of `.false.` indicates that there are still unfinished operations that require additional calls with `routesyncflag = ESMF_ROUTESYNC_NBTESTFINISH`, or a final call with `routesyncflag = ESMF_ROUTESYNC_NBWAITFINISH`. For all other `routesyncflag` settings the returned value in `finishedflag` is always `.true.`.

[cancelledflag] A value of `.true.` indicates that were cancelled communication operations. In this case the data in the `dstArray` must be considered invalid. It may have been partially modified by the call. A value of `.false.` indicates that none of the communication operations was cancelled. The data in `dstArray` is valid if `finishedflag` returns equal `.true.`.

[checkflag] If set to `.TRUE.` the input Array pair will be checked for consistency with the precomputed operation provided by `routehandle`. If set to `.FALSE.` *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set `checkflag` to `.FALSE.` to achieve highest performance.

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

28.5.23 ESMF_ArrayHaloRelease - Release resources associated with Array halo operation

INTERFACE:

```
subroutine ESMF_ArrayHaloRelease(routehandle, noGarbage, rc)
```

ARGUMENTS:

```
 type(ESMF_RouteHandle), intent(inout)         :: routehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
 logical,                intent(in),  optional :: noGarbage
 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:

  8.0.0 Added argument `noGarbage`. The argument provides a mechanism to override the default garbage collection mechanism when destroying an ESMF object.

DESCRIPTION:

Release resources associated with an Array halo operation. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

[noGarbage] If set to `.TRUE.` the object will be fully destroyed and removed from the ESMF garbage collection system. Note however that under this condition ESMF cannot protect against accessing the destroyed object through dangling aliases – a situation which may lead to hard to debug application crashes.

It is generally recommended to leave the `noGarbage` argument set to `.FALSE.` (the default), and to take advantage of the ESMF garbage collection system which will prevent problems with dangling aliases or incorrect sequences of destroy calls. However this level of support requires that a small remnant of the object is kept in memory past the destroy call. This can lead to an unexpected increase in memory consumption over the course of execution in applications that use temporary ESMF objects. For situations where the repeated creation and destruction of temporary objects leads to memory issues, it is recommended to call with `noGarbage` set to `.TRUE.`, fully removing the entire temporary object from memory.

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

28.5.24 ESMF_ArrayHaloStore - Precompute an Array halo operation

INTERFACE:

```
subroutine ESMF_ArrayHaloStore(array, routehandle, &
  startregion, haloLDepth, haloUDepth, pipelineDepth, rc)
```

ARGUMENTS:

```
type(ESMF_Array),            intent(inout)           :: array
type(ESMF_RouteHandle),      intent(inout)           :: routehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_StartRegion_Flag), intent(in),    optional :: startregion
integer,                     intent(in),    optional :: haloLDepth(:)
integer,                     intent(in),    optional :: haloUDepth(:)
integer,                     intent(inout), optional :: pipelineDepth
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:

  6.1.0 Added argument `pipelineDepth`. The new argument provide access to the tuning parameter affecting the sparse matrix execution.

DESCRIPTION:

Store an Array halo operation over the data in `array`. By default, i.e. without specifying `startregion`, `haloLDepth` and `haloUDepth`, all elements in the total Array region that lie outside the exclusive region will

be considered potential destination elements for halo. However, only those elements that have a corresponding halo source element, i.e. an exclusive element on one of the DEs, will be updated under the halo operation. Elements that have no associated source remain unchanged under halo.

Specifying `startregion` allows the shape of the effective halo region to be changed from the inside. Setting this flag to `ESMF_STARTREGION_COMPUTATIONAL` means that only elements outside the computational region of the Array are considered for potential destination elements for the halo operation. The default is `ESMF_STARTREGION_EXCLUSIVE`.

The `haloLDepth` and `haloUDepth` arguments allow to reduce the extent of the effective halo region. Starting at the region specified by `startregion`, the `haloLDepth` and `haloUDepth` define a halo depth in each direction. Note that the maximum halo region is limited by the total Array region, independent of the actual `haloLDepth` and `haloUDepth` setting. The total Array region is local DE specific. The `haloLDepth` and `haloUDepth` are interpreted as the maximum desired extent, reducing the potentially larger region available for the halo operation.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_ArrayHalo()` on any Array that matches `array` in *type*, *kind*, and memory layout of the *distributed* dimensions. However, the size, number, and index order of *undistributed* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This call is *collective* across the current VM.

array `ESMF_Array` containing data to be haloed. The data in the halo region may be destroyed by this call.

routehandle Handle to the precomputed Route.

[startregion] The start of the effective halo region on every DE. The default setting is `ESMF_STARTREGION_EXCLUSIVE`, rendering all non-exclusive elements potential halo destination elements. See section **??** for a complete list of valid settings.

[haloLDepth] This vector specifies the lower corner of the effective halo region with respect to the lower corner of `startregion`. The size of `haloLDepth` must equal the number of distributed Array dimensions.

[haloUDepth] This vector specifies the upper corner of the effective halo region with respect to the upper corner of `startregion`. The size of `haloUDepth` must equal the number of distributed Array dimensions.

[pipelineDepth] The `pipelineDepth` parameter controls how many messages a PET may have outstanding during a halo exchange. Larger values of `pipelineDepth` typically lead to better performance. However, on some systems too large a value may lead to performance degradation, or runtime errors.

Note that the pipeline depth has no effect on the bit-for-bit reproducibility of the results. However, it may affect the performance reproducibility of the exchange.

The `ESMF_ArraySMMStore()` method implements an auto-tuning scheme for the `pipelineDepth` parameter. The intent on the `pipelineDepth` argument is `"inout"` in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `pipelineDepth` parameter, and the auto-tuning phase is skipped. In this case the `pipelineDepth` argument is not modified on return. If the provided argument is < 0, the `pipelineDepth` parameter is determined internally using the auto-tuning scheme. In this case the `pipelineDepth` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `pipelineDepth` argument is omitted.

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

28.5.25 ESMF_ArrayIsCreated - Check whether an Array object has been created

INTERFACE:

```
   function ESMF_ArrayIsCreated(array, rc)
```

RETURN VALUE:

```
     logical :: ESMF_ArrayIsCreated
```

ARGUMENTS:

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

DESCRIPTION:

Return `.true.` if the `array` has been created. Otherwise return `.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:

array `ESMF_Array` queried.

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

28.5.26 ESMF_ArrayPrint - Print Array information

INTERFACE:

```
   subroutine ESMF_ArrayPrint(array, rc)
```

ARGUMENTS:

```
     type(ESMF_Array), intent(in)            :: array
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     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:

Print internal information of the specified `ESMF_Array` object.

The arguments are:

array ESMF_Array object.

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

28.5.27 ESMF_ArrayRead - Read Array data from a file

INTERFACE:

```
   subroutine ESMF_ArrayRead(array, fileName, variableName, &
     timeslice, iofmt, rc)
     ! We need to terminate the strings on the way to C++
```

ARGUMENTS:

```
     type(ESMF_Array),      intent(inout)         :: array
     character(*),          intent(in)            :: fileName
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     character(*),          intent(in),  optional :: variableName
     integer,               intent(in),  optional :: timeslice
     type(ESMF_IOFmt_Flag), intent(in),  optional :: iofmt
     integer,               intent(out), optional :: rc
```

DESCRIPTION:

Read Array data from file and put it into an ESMF_Array object. For this API to be functional, the environment variable ESMF_PIO should be set to "internal" when the ESMF library is built. Please see the section on Data I/O, **??**.

Limitations:

- Only single tile Arrays are supported.
- Not supported in ESMF_COMM=mpiuni mode.

The arguments are:

array The ESMF_Array object in which the read data is returned.

fileName The name of the file from which Array data is read.

[variableName] Variable name in the file; default is the "name" of Array. Use this argument only in the I/O format (such as NetCDF) that supports variable name. If the I/O format does not support this (such as binary format), ESMF will return an error code.

[timeslice] The time-slice number of the variable read from file.

[iofmt] The I/O format. Please see Section **??** for the list of options. If not present, file names with a .bin extension will use ESMF_IOFMT_BIN, and file names with a .nc extension will use ESMF_IOFMT_NETCDF. Other files default to ESMF_IOFMT_NETCDF.

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

28.5.28 ESMF_ArrayRedist - Execute an Array redistribution

INTERFACE:

```
   subroutine ESMF_ArrayRedist(srcArray, dstArray, routehandle, &
     routesyncflag, finishedflag, cancelledflag, zeroregion, checkflag, rc)
```

ARGUMENTS:

```
     type(ESMF_Array),          intent(in),    optional :: srcArray
     type(ESMF_Array),          intent(inout), optional :: dstArray
     type(ESMF_RouteHandle),    intent(inout)           :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_RouteSync_Flag), intent(in),    optional :: routesyncflag
     logical,                   intent(out),   optional :: finishedflag
     logical,                   intent(out),   optional :: cancelledflag
     type(ESMF_Region_Flag),    intent(in),    optional :: zeroregion
     logical,                   intent(in),    optional :: checkflag
     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:

  7.1.0r Added argument `zeroregion` to allow user to control how the destination array is zero'ed out. This is especially useful in cases where the source and destination arrays do not cover the identical index space.

DESCRIPTION:

Execute a precomputed Array redistribution from `srcArray` to `dstArray`. Both `srcArray` and `dstArray` must match the respective Arrays used during `ESMF_ArrayRedisttore()` in *type*, *kind*, and memory layout of the *distributed* dimensions. However, the size, number, and index order of *undistributed* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

The `srcArray` and `dstArray` arguments are optional in support of the situation where `srcArray` and/or `dstArray` are not defined on all PETs. The `srcArray` and `dstArray` must be specified on those PETs that hold source or destination DEs, respectively, but may be omitted on all other PETs. PETs that hold neither source nor destination DEs may omit both arguments.

It is erroneous to specify the identical Array object for `srcArray` and `dstArray` arguments.

See `ESMF_ArrayRedistStore()` on how to precompute `routehandle`.

This call is *collective* across the current VM.

[srcArray] `ESMF_Array` with source data.

[dstArray] ESMF_Array with destination data.

routehandle Handle to the precomputed Route.

[routesyncflag] Indicate communication option. Default is ESMF_ROUTESYNC_BLOCKING, resulting in a blocking operation. See section **??** for a complete list of valid settings.

[finishedflag] Used in combination with routesyncflag = ESMF_ROUTESYNC_NBTESTFINISH. Returned finishedflag equal to .true. indicates that all operations have finished. A value of .false. indicates that there are still unfinished operations that require additional calls with routesyncflag = ESMF_ROUTESYNC_NBTESTFINISH, or a final call with routesyncflag = ESMF_ROUTESYNC_NBWAITFINISH. For all other routesyncflag settings the returned value in finishedflag is always .true..

[cancelledflag] A value of .true. indicates that were cancelled communication operations. In this case the data in the dstArray must be considered invalid. It may have been partially modified by the call. A value of .false. indicates that none of the communication operations was cancelled. The data in dstArray is valid if finishedflag returns equal .true..

[zeroregion] If set to ESMF_REGION_TOTAL the total regions of all DEs in dstArray will be initialized to zero before updating the elements with the results of the sparse matrix multiplication. If set to ESMF_REGION_EMPTY the elements in dstArray will not be modified prior to the sparse matrix multiplication and results will be added to the incoming element values. Setting zeroregion to ESMF_REGION_SELECT will only zero out those elements in the destination Array that will be updated by the sparse matrix multiplication. See section **??** for a complete list of valid settings. The default is ESMF_REGION_SELECT.

[checkflag] If set to .TRUE. the input Array pair will be checked for consistency with the precomputed operation provided by routehandle. If set to .FALSE. *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set checkflag to .FALSE. to achieve highest performance.

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

28.5.29 ESMF_ArrayRedistRelease - Release resources associated with Array redistribution

INTERFACE:

```
subroutine ESMF_ArrayRedistRelease(routehandle, noGarbage, rc)
```

ARGUMENTS:

```
 type(ESMF_RouteHandle), intent(inout)         :: routehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
 logical,                intent(in),  optional :: noGarbage
 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:

  8.0.0 Added argument `noGarbage`. The argument provides a mechanism to override the default garbage collection mechanism when destroying an ESMF object.

DESCRIPTION:

Release resources associated with an Array redistribution. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

[noGarbage] If set to `.TRUE.` the object will be fully destroyed and removed from the ESMF garbage collection system. Note however that under this condition ESMF cannot protect against accessing the destroyed object through dangling aliases – a situation which may lead to hard to debug application crashes.

It is generally recommended to leave the `noGarbage` argument set to `.FALSE.` (the default), and to take advantage of the ESMF garbage collection system which will prevent problems with dangling aliases or incorrect sequences of destroy calls. However this level of support requires that a small remnant of the object is kept in memory past the destroy call. This can lead to an unexpected increase in memory consumption over the course of execution in applications that use temporary ESMF objects. For situations where the repeated creation and destruction of temporary objects leads to memory issues, it is recommended to call with `noGarbage` set to `.TRUE.`, fully removing the entire temporary object from memory.

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

28.5.30 ESMF_ArrayRedistStore - Precompute Array redistribution with local factor argument

INTERFACE:

```
! Private name; call using ESMF_ArrayRedistStore()
subroutine ESMF_ArrayRedistStore<type><kind>(srcArray, dstArray, &
  routehandle, factor, srcToDstTransposeMap, &
  ignoreUnmatchedIndices, pipelineDepth, rc)
```

ARGUMENTS:

```
  type(ESMF_Array),       intent(in)              :: srcArray
  type(ESMF_Array),       intent(inout)           :: dstArray
  type(ESMF_RouteHandle), intent(inout)           :: routehandle
  <type>(ESMF_KIND_<kind>),intent(in)             :: factor
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  integer,                intent(in),    optional :: srcToDstTransposeMap(:)
  logical,                intent(in),    optional :: ignoreUnmatchedIndices
  integer,                intent(inout), optional :: pipelineDepth
  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:

  6.1.0 Added argument `pipelineDepth`. The new argument provide access to the tuning parameter affecting the sparse matrix execution.

  7.0.0 Added argument `transposeRoutehandle` to allow a handle to the transposed redist operation to be returned.
  Added argument `ignoreUnmatchedIndices` to support situations where not all elements between source and destination Arrays match.

  7.1.0r Removed argument `transposeRoutehandle` and provide it via interface overloading instead. This allows argument `srcArray` to stay strictly intent(in) for this entry point.

DESCRIPTION:

`ESMF_ArrayRedistStore()` is a collective method across all PETs of the current Component. The interface of the method is overloaded, allowing – in principle – each PET to call into `ESMF_ArrayRedistStore()` through a different entry point. Restrictions apply as to which combinations are sensible. All other combinations result in ESMF run time errors. The complete semantics of the `ESMF_ArrayRedistStore()` method, as provided through the separate entry points shown in 28.5.30 and 28.5.32, is described in the following paragraphs as a whole.

Store an Array redistribution operation from `srcArray` to `dstArray`. Interface 28.5.30 allows PETs to specify a `factor` argument. PETs not specifying a `factor` argument call into interface 28.5.32. If multiple PETs specify the `factor` argument, its type and kind, as well as its value must match across all PETs. If none of the PETs specify a `factor` argument the default will be a factor of 1. The resulting factor is applied to all of the source data during redistribution, allowing scaling of the data, e.g. for unit transformation.

Both `srcArray` and `dstArray` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 28.2.18 for details on the definition of *sequence indices*.

Source Array, destination Array, and the factor may be of different <type><kind>. Further, source and destination Arrays may differ in shape, however, the number of elements must match.

If `srcToDstTransposeMap` is not specified the redistribution corresponds to an identity mapping of the sequentialized source Array to the sequentialized destination Array. If the `srcToDstTransposeMap` argument is provided it must be identical on all PETs. The `srcToDstTransposeMap` allows source and destination Array dimensions to be transposed during the redistribution. The number of source and destination Array dimensions must be equal under this condition and the size of mapped dimensions must match.

It is erroneous to specify the identical Array object for `srcArray` and `dstArray` arguments.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_ArrayRedist()` on any pair of Arrays that matches `srcArray` and `dstArray` in *type*, *kind*, and memory layout of the *distributed* dimensions. However, the size, number, and index order of *undistributed* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This method is overloaded for:
`ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`,
`ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

This call is *collective* across the current VM.

srcArray ESMF_Array with source data.

dstArray ESMF_Array with destination data. The data in this Array may be destroyed by this call.

routehandle Handle to the precomputed Route.

factor Factor by which to multiply source data.

[srcToDstTransposeMap] List with as many entries as there are dimensions in srcArray. Each entry maps the corresponding srcArray dimension against the specified dstArray dimension. Mixing of distributed and undistributed dimensions is supported.

[ignoreUnmatchedIndices] A logical flag that affects the behavior for when not all elements match between the srcArray and dstArray side. The default setting is .false., indicating that it is an error when such a situation is encountered. Setting ignoreUnmatchedIndices to .true. ignores unmatched indices.

[pipelineDepth] The pipelineDepth parameter controls how many messages a PET may have outstanding during a redist exchange. Larger values of pipelineDepth typically lead to better performance. However, on some systems too large a value may lead to performance degradation, or runtime errors.

Note that the pipeline depth has no effect on the bit-for-bit reproducibility of the results. However, it may affect the performance reproducibility of the exchange.

The ESMF_ArraySMMStore() method implements an auto-tuning scheme for the pipelineDepth parameter. The intent on the pipelineDepth argument is "inout" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the pipelineDepth parameter, and the auto-tuning phase is skipped. In this case the pipelineDepth argument is not modified on return. If the provided argument is < 0, the pipelineDepth parameter is determined internally using the auto-tuning scheme. In this case the pipelineDepth argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional pipelineDepth argument is omitted.

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

28.5.31 ESMF_ArrayRedistStore - Precompute Array redistribution and transpose with local factor argument

INTERFACE:

```
! Private name; call using ESMF_ArrayRedistStore()
subroutine ESMF_ArrayRedistStore<type><kind>TP(srcArray, dstArray, &
  routehandle, transposeRoutehandle, factor, &
  srcToDstTransposeMap, ignoreUnmatchedIndices, pipelineDepth, rc)
```

ARGUMENTS:

```
  type(ESMF_Array),       intent(inout)           :: srcArray
  type(ESMF_Array),       intent(inout)           :: dstArray
  type(ESMF_RouteHandle), intent(inout)           :: routehandle
  type(ESMF_RouteHandle), intent(inout)           :: transposeRoutehandle
  <type>(ESMF_KIND_<kind>),intent(in)             :: factor
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
```

```
    integer,                intent(in),    optional :: srcToDstTransposeMap(:)
    logical,                intent(in),    optional :: ignoreUnmatchedIndices
    integer,                intent(inout), optional :: pipelineDepth
    integer,                intent(out),   optional :: rc
```

DESCRIPTION:

ESMF_ArrayRedistStore() is a collective method across all PETs of the current Component. The interface of the method is overloaded, allowing – in principle – each PET to call into ESMF_ArrayRedistStore() through a different entry point. Restrictions apply as to which combinations are sensible. All other combinations result in ESMF run time errors. The complete semantics of the ESMF_ArrayRedistStore() method, as provided through the separate entry points shown in 28.5.31 and 28.5.33, is described in the following paragraphs as a whole.

Store an Array redistribution operation from srcArray to dstArray. Interface 28.5.31 allows PETs to specify a factor argument. PETs not specifying a factor argument call into interface 28.5.33. If multiple PETs specify the factor argument, its type and kind, as well as its value must match across all PETs. If none of the PETs specify a factor argument the default will be a factor of 1. The resulting factor is applied to all of the source data during redistribution, allowing scaling of the data, e.g. for unit transformation.

Both srcArray and dstArray are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 28.2.18 for details on the definition of *sequence indices*.

Source Array, destination Array, and the factor may be of different <type><kind>. Further, source and destination Arrays may differ in shape, however, the number of elements must match.

If srcToDstTransposeMap is not specified the redistribution corresponds to an identity mapping of the sequentialized source Array to the sequentialized destination Array. If the srcToDstTransposeMap argument is provided it must be identical on all PETs. The srcToDstTransposeMap allows source and destination Array dimensions to be transposed during the redistribution. The number of source and destination Array dimensions must be equal under this condition and the size of mapped dimensions must match.

It is erroneous to specify the identical Array object for srcArray and dstArray arguments.

The routine returns an ESMF_RouteHandle that can be used to call ESMF_ArrayRedist() on any pair of Arrays that matches srcArray and dstArray in *type*, *kind*, and memory layout of the *distributed* dimensions. However, the size, number, and index order of *undistributed* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This method is overloaded for:
ESMF_TYPEKIND_I4, ESMF_TYPEKIND_I8,
ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8.

This call is *collective* across the current VM.

srcArray ESMF_Array with source data. The data in this Array may be destroyed by this call.

dstArray ESMF_Array with destination data. The data in this Array may be destroyed by this call.

routehandle Handle to the precomputed Route.

transposeRoutehandle Handle to the transposed matrix operation. The transposed operation goes from dstArray to srcArray.

factor Factor by which to multiply source data.

[srcToDstTransposeMap] List with as many entries as there are dimensions in srcArray. Each entry maps the corresponding srcArray dimension against the specified dstArray dimension. Mixing of distributed and undistributed dimensions is supported.

[ignoreUnmatchedIndices] A logical flag that affects the behavior for when not all elements match between the `srcArray` and `dstArray` side. The default setting is `.false.`, indicating that it is an error when such a situation is encountered. Setting `ignoreUnmatchedIndices` to `.true.` ignores unmatched indices.

[pipelineDepth] The `pipelineDepth` parameter controls how many messages a PET may have outstanding during a redist exchange. Larger values of `pipelineDepth` typically lead to better performance. However, on some systems too large a value may lead to performance degradation, or runtime errors.

Note that the pipeline depth has no effect on the bit-for-bit reproducibility of the results. However, it may affect the performance reproducibility of the exchange.

The `ESMF_ArraySMMStore()` method implements an auto-tuning scheme for the `pipelineDepth` parameter. The intent on the `pipelineDepth` argument is "`inout`" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `pipelineDepth` parameter, and the auto-tuning phase is skipped. In this case the `pipelineDepth` argument is not modified on return. If the provided argument is < 0, the `pipelineDepth` parameter is determined internally using the auto-tuning scheme. In this case the `pipelineDepth` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `pipelineDepth` argument is omitted.

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

28.5.32 ESMF_ArrayRedistStore - Precompute Array redistribution without local factor argument

INTERFACE:

```
! Private name; call using ESMF_ArrayRedistStore()
subroutine ESMF_ArrayRedistStoreNF(srcArray, dstArray, routehandle, &
  srcToDstTransposeMap, ignoreUnmatchedIndices, &
  pipelineDepth, rc)
```

ARGUMENTS:

```
  type(ESMF_Array),       intent(in)              :: srcArray
  type(ESMF_Array),       intent(inout)           :: dstArray
  type(ESMF_RouteHandle), intent(inout)           :: routehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  integer,                intent(in),    optional :: srcToDstTransposeMap(:)
  logical,                intent(in),    optional :: ignoreUnmatchedIndices
  integer,                intent(inout), optional :: pipelineDepth
  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:

6.1.0 Added argument `pipelineDepth`. The new argument provide access to the tuning parameter affecting the sparse matrix execution.

7.0.0 Added argument `transposeRoutehandle` to allow a handle to the transposed redist operation to be returned.
Added argument `ignoreUnmatchedIndices` to support situations where not all elements between source and destination Arrays match.

7.1.0r Removed argument `transposeRoutehandle` and provide it via interface overloading instead. This allows argument `srcArray` to stay strictly intent(in) for this entry point.

DESCRIPTION:

`ESMF_ArrayRedistStore()` is a collective method across all PETs of the current Component. The interface of the method is overloaded, allowing – in principle – each PET to call into `ESMF_ArrayRedistStore()` through a different entry point. Restrictions apply as to which combinations are sensible. All other combinations result in ESMF run time errors. The complete semantics of the `ESMF_ArrayRedistStore()` method, as provided through the separate entry points shown in 28.5.30 and 28.5.32, is described in the following paragraphs as a whole.

Store an Array redistribution operation from `srcArray` to `dstArray`. Interface 28.5.30 allows PETs to specify a `factor` argument. PETs not specifying a `factor` argument call into interface 28.5.32. If multiple PETs specify the `factor` argument, its type and kind, as well as its value must match across all PETs. If none of the PETs specify a `factor` argument the default will be a factor of 1. The resulting factor is applied to all of the source data during redistribution, allowing scaling of the data, e.g. for unit transformation.

Both `srcArray` and `dstArray` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 28.2.18 for details on the definition of *sequence indices*.

Source Array, destination Array, and the factor may be of different <type><kind>. Further, source and destination Arrays may differ in shape, however, the number of elements must match.

If `srcToDstTransposeMap` is not specified the redistribution corresponds to an identity mapping of the sequentialized source Array to the sequentialized destination Array. If the `srcToDstTransposeMap` argument is provided it must be identical on all PETs. The `srcToDstTransposeMap` allows source and destination Array dimensions to be transposed during the redistribution. The number of source and destination Array dimensions must be equal under this condition and the size of mapped dimensions must match.

It is erroneous to specify the identical Array object for `srcArray` and `dstArray` arguments.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_ArrayRedist()` on any pair of Arrays that matches `srcArray` and `dstArray` in *type*, *kind*, and memory layout of the *distributed* dimensions. However, the size, number, and index order of *undistributed* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This call is *collective* across the current VM.

srcArray `ESMF_Array` with source data.

dstArray `ESMF_Array` with destination data. The data in this Array may be destroyed by this call.

routehandle Handle to the precomputed Route.

[srcToDstTransposeMap] List with as many entries as there are dimensions in `srcArray`. Each entry maps the corresponding `srcArray` dimension against the specified `dstArray` dimension. Mixing of distributed and undistributed dimensions is supported.

[ignoreUnmatchedIndices] A logical flag that affects the behavior for when not all elements match between the `srcArray` and `dstArray` side. The default setting is `.false.`, indicating that it is an error when such a situation is encountered. Setting `ignoreUnmatchedIndices` to `.true.` ignores unmatched indices.

[pipelineDepth] The `pipelineDepth` parameter controls how many messages a PET may have outstanding during a redist exchange. Larger values of `pipelineDepth` typically lead to better performance. However, on some systems too large a value may lead to performance degradation, or runtime errors.

Note that the pipeline depth has no effect on the bit-for-bit reproducibility of the results. However, it may affect the performance reproducibility of the exchange.

The `ESMF_ArraySMMStore()` method implements an auto-tuning scheme for the `pipelineDepth` parameter. The intent on the `pipelineDepth` argument is "`inout`" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `pipelineDepth` parameter, and the auto-tuning phase is skipped. In this case the `pipelineDepth` argument is not modified on return. If the provided argument is < 0, the `pipelineDepth` parameter is determined internally using the auto-tuning scheme. In this case the `pipelineDepth` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `pipelineDepth` argument is omitted.

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

28.5.33 ESMF_ArrayRedistStore - Precompute Array redistribution and transpose without local factor argument

INTERFACE:

```
! Private name; call using ESMF_ArrayRedistStore()
subroutine ESMF_ArrayRedistStoreNFTP(srcArray, dstArray, routehandle, &
  transposeRoutehandle, srcToDstTransposeMap, &
  ignoreUnmatchedIndices, pipelineDepth, rc)
```

ARGUMENTS:

```
  type(ESMF_Array),       intent(inout)           :: srcArray
  type(ESMF_Array),       intent(inout)           :: dstArray
  type(ESMF_RouteHandle), intent(inout)           :: routehandle
  type(ESMF_RouteHandle), intent(inout)           :: transposeRoutehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  integer,                intent(in),    optional :: srcToDstTransposeMap(:)
  logical,                intent(in),    optional :: ignoreUnmatchedIndices
  integer,                intent(inout), optional :: pipelineDepth
  integer,                intent(out),   optional :: rc
```

DESCRIPTION:

`ESMF_ArrayRedistStore()` is a collective method across all PETs of the current Component. The interface of the method is overloaded, allowing – in principle – each PET to call into `ESMF_ArrayRedistStore()` through a different entry point. Restrictions apply as to which combinations are sensible. All other combinations result in ESMF run time errors. The complete semantics of the `ESMF_ArrayRedistStore()` method, as provided through the separate entry points shown in 28.5.31 and 28.5.33, is described in the following paragraphs as a whole.

Store an Array redistribution operation from `srcArray` to `dstArray`. Interface 28.5.31 allows PETs to specify a `factor` argument. PETs not specifying a `factor` argument call into interface 28.5.33. If multiple PETs specify the `factor` argument, its type and kind, as well as its value must match across all PETs. If none of the PETs specify

a factor argument the default will be a factor of 1. The resulting factor is applied to all of the source data during redistribution, allowing scaling of the data, e.g. for unit transformation.

Both srcArray and dstArray are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 28.2.18 for details on the definition of *sequence indices*.

Source Array, destination Array, and the factor may be of different <type><kind>. Further, source and destination Arrays may differ in shape, however, the number of elements must match.

If srcToDstTransposeMap is not specified the redistribution corresponds to an identity mapping of the sequentialized source Array to the sequentialized destination Array. If the srcToDstTransposeMap argument is provided it must be identical on all PETs. The srcToDstTransposeMap allows source and destination Array dimensions to be transposed during the redistribution. The number of source and destination Array dimensions must be equal under this condition and the size of mapped dimensions must match.

It is erroneous to specify the identical Array object for srcArray and dstArray arguments.

The routine returns an ESMF_RouteHandle that can be used to call ESMF_ArrayRedist() on any pair of Arrays that matches srcArray and dstArray in *type*, *kind*, and memory layout of the *distributed* dimensions. However, the size, number, and index order of *undistributed* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This call is *collective* across the current VM.

srcArray ESMF_Array with source data. The data in this Array may be destroyed by this call.

dstArray ESMF_Array with destination data. The data in this Array may be destroyed by this call.

routehandle Handle to the precomputed Route.

transposeRoutehandle Handle to the transposed matrix operation. The transposed operation goes from dstArray to srcArray.

[srcToDstTransposeMap] List with as many entries as there are dimensions in srcArray. Each entry maps the corresponding srcArray dimension against the specified dstArray dimension. Mixing of distributed and undistributed dimensions is supported.

[ignoreUnmatchedIndices] A logical flag that affects the behavior for when not all elements match between the srcArray and dstArray side. The default setting is .false., indicating that it is an error when such a situation is encountered. Setting ignoreUnmatchedIndices to .true. ignores unmatched indices.

[pipelineDepth] The pipelineDepth parameter controls how many messages a PET may have outstanding during a redist exchange. Larger values of pipelineDepth typically lead to better performance. However, on some systems too large a value may lead to performance degradation, or runtime errors.

Note that the pipeline depth has no effect on the bit-for-bit reproducibility of the results. However, it may affect the performance reproducibility of the exchange.

The ESMF_ArraySMMStore() method implements an auto-tuning scheme for the pipelineDepth parameter. The intent on the pipelineDepth argument is "inout" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the pipelineDepth parameter, and the auto-tuning phase is skipped. In this case the pipelineDepth argument is not modified on return. If the provided argument is < 0, the pipelineDepth parameter is determined internally using the auto-tuning scheme. In this case the pipelineDepth argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional pipelineDepth argument is omitted.

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

28.5.34 ESMF_ArrayScatter - Scatter a Fortran array across the ESMF_Array

INTERFACE:

```
subroutine ESMF_ArrayScatter(array, farray, rootPet, tile, vm, rc)
```

ARGUMENTS:

```
type(ESMF_Array), intent(inout) :: array
<type> (ESMF_KIND_<kind>), intent(in), target :: farray(<rank>)
integer, intent(in) :: rootPet
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
integer, intent(in), optional :: tile
type(ESMF_VM), intent(in), 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.

DESCRIPTION:

Scatter the data of `farray` located on `rootPET` across an ESMF_Array object. A single `farray` must be scattered across a single DistGrid tile in Array. The optional `tile` argument allows selection of the tile. For Arrays defined on a single tile DistGrid the default selection (tile 1) will be correct. The shape of `farray` must match the shape of the tile in Array.

If the Array contains replicating DistGrid dimensions data will be scattered across all of the replicated pieces.

This version of the interface implements the PET-based blocking paradigm: Each PET of the VM must issue this call exactly once for *all* of its DEs. The call will block until all PET-local data objects are accessible.

The arguments are:

array The `ESMF_Array` object across which data will be scattered.

{farray} The Fortran array that is to be scattered. Only root must provide a valid `farray`, the other PETs may treat `farray` as an optional argument.

rootPet PET that holds the valid data in `farray`.

[tile] The DistGrid tile in `array` into which to scatter `farray`. By default `farray` will be scattered into tile 1.

[vm] Optional `ESMF_VM` object of the current context. Providing the VM of the current context will lower the method's overhead.

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

28.5.35 ESMF_ArraySet - Set object-wide Array information

INTERFACE:

```
! Private name; call using ESMF_ArraySet()
subroutine ESMF_ArraySetDefault(array, computationalLWidth, &
  computationalUWidth, name, rc)
```

ARGUMENTS:

```
     type(ESMF_Array),   intent(inout)         :: array
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,            intent(in),  optional :: computationalLWidth(:,:)
     integer,            intent(in),  optional :: computationalUWidth(:,:)
     character(len = *), intent(in),  optional :: name
     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:

Sets adjustable settings in an `ESMF_Array` object. Arrays with tensor dimensions will set values for *all* tensor components.

The arguments are:

array `ESMF_Array` object for which to set properties.

[computationalLWidth] This argument must have of size `(dimCount, localDeCount)`. `computationalLWidth` specifies the lower corner of the computational region with respect to the lower corner of the exclusive region for all local DEs.

[computationalUWidth] This argument must have of size `(dimCount, localDeCount)`. `computationalUWidth` specifies the upper corner of the computational region with respect to the upper corner of the exclusive region for all local DEs.

[name] The Array name.

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

28.5.36 ESMF_ArraySet - Set DE-local Array information

INTERFACE:

```
   ! Private name; call using ESMF_ArraySet()
   subroutine ESMF_ArraySetPLocalDe(array, localDe, rimSeqIndex, rc)
```

ARGUMENTS:

```
     type(ESMF_Array),   intent(inout)         :: array
     integer,            intent(in)            :: localDe
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,            intent(in),  optional :: rimSeqIndex(:)
     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:

Sets adjustable settings in an `ESMF_Array` object for a specific localDe.

The arguments are:

array `ESMF_Array` object for which to set properties.

localDe Local DE for which to set values.

[rimSeqIndex] Sequence indices in the halo rim of localDe.

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

28.5.37 ESMF_ArraySMM - Execute an Array sparse matrix multiplication

INTERFACE:

```
   subroutine ESMF_ArraySMM(srcArray, dstArray, routehandle, &
     routesyncflag, finishedflag, cancelledflag, zeroregion, termorderflag, &
     checkflag, dynamicMask, rc)
```

ARGUMENTS:

```
     type(ESMF_Array),               intent(in),    optional :: srcArray
     type(ESMF_Array),               intent(inout), optional :: dstArray
     type(ESMF_RouteHandle),         intent(inout)           :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     type(ESMF_RouteSync_Flag),      intent(in),    optional :: routesyncflag
     logical,                        intent(out),   optional :: finishedflag
     logical,                        intent(out),   optional :: cancelledflag
     type(ESMF_Region_Flag),         intent(in),    optional :: zeroregion
```

```
type(ESMF_TermOrder_Flag),      intent(in),    optional :: termorderflag
logical,                        intent(in),    optional :: checkflag
type(ESMF_DynamicMask), target, intent(in),    optional :: dynamicMask
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:

  6.1.0 Added argument `termorderflag`. The new argument gives the user control over the order in which the src terms are summed up.

  7.1.0r Added argument `dynamicMask`. The new argument supports the dynamic masking feature.

DESCRIPTION:

Execute a precomputed Array sparse matrix multiplication from `srcArray` to `dstArray`. Both `srcArray` and `dstArray` must match the respective Arrays used during `ESMF_ArraySMMStore()` in *type*, *kind*, and memory layout of the *distributed* dimensions. However, the size, number, and index order of *undistributed* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

The `srcArray` and `dstArray` arguments are optional in support of the situation where `srcArray` and/or `dstArray` are not defined on all PETs. The `srcArray` and `dstArray` must be specified on those PETs that hold source or destination DEs, respectively, but may be omitted on all other PETs. PETs that hold neither source nor destination DEs may omit both arguments.

It is erroneous to specify the identical Array object for `srcArray` and `dstArray` arguments.

See `ESMF_ArraySMMStore()` on how to precompute `routehandle`. See section 28.2.18 for details on the operation `ESMF_ArraySMM()` performs.

This call is *collective* across the current VM.

[srcArray] `ESMF_Array` with source data.

[dstArray] `ESMF_Array` with destination data.

routehandle Handle to the precomputed Route.

[routesyncflag] Indicate communication option. Default is `ESMF_ROUTESYNC_BLOCKING`, resulting in a blocking operation. See section **??** for a complete list of valid settings.

[finishedflag] Used in combination with `routesyncflag = ESMF_ROUTESYNC_NBTESTFINISH`. Returned `finishedflag` equal to `.true.` indicates that all operations have finished. A value of `.false.` indicates that there are still unfinished operations that require additional calls with `routesyncflag = ESMF_ROUTESYNC_NBTESTFINISH`, or a final call with `routesyncflag = ESMF_ROUTESYNC_NBWAITFINISH`. For all other `routesyncflag` settings the returned value in `finishedflag` is always `.true.`.

[cancelledflag] A value of `.true.` indicates that were cancelled communication operations. In this case the data in the `dstArray` must be considered invalid. It may have been partially modified by the call. A value of `.false.` indicates that none of the communication operations was cancelled. The data in `dstArray` is valid if `finishedflag` returns equal `.true.`.

[zeroregion] If set to ESMF_REGION_TOTAL *(default)* the total regions of all DEs in dstArray will be initialized to zero before updating the elements with the results of the sparse matrix multiplication. If set to ESMF_REGION_EMPTY the elements in dstArray will not be modified prior to the sparse matrix multiplication and results will be added to the incoming element values. Setting zeroregion to ESMF_REGION_SELECT will only zero out those elements in the destination Array that will be updated by the sparse matrix multiplication. See section **??** for a complete list of valid settings.

[termorderflag] Specifies the order of the source side terms in all of the destination sums. The termorderflag only affects the order of terms during the execution of the RouteHandle. See the **??** section for an in-depth discussion of *all* bit-for-bit reproducibility aspects related to route-based communication methods. See **??** for a full list of options. The default setting depends on whether the dynamicMask argument is present or not. With dynamicMask argument present, the default of termorderflag is ESMF_TERMORDER_SRCSEQ. This ensures that all source terms are present on the destination side, and the interpolation can be calculated as a single sum. When dynamicMask is absent, the default of termorderflag is ESMF_TERMORDER_FREE, allowing maximum flexibility and partial sums for optimum performance.

[checkflag] If set to .TRUE. the input Array pair will be checked for consistency with the precomputed operation provided by routehandle. If set to .FALSE. *(default)* only a very basic input check will be performed, leaving many inconsistencies undetected. Set checkflag to .FALSE. to achieve highest performance.

[dynamicMask] Object holding dynamic masking information. See section **??** for a discussion of dynamic masking.

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

28.5.38 ESMF_ArraySMMRelease - Release resources associated with Array sparse matrix multiplication

INTERFACE:

```
subroutine ESMF_ArraySMMRelease(routehandle, noGarbage, rc)
```

ARGUMENTS:

```
 type(ESMF_RouteHandle), intent(inout)         :: routehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
 logical,                intent(in),  optional :: noGarbage
 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:

  8.0.0 Added argument noGarbage. The argument provides a mechanism to override the default garbage collection mechanism when destroying an ESMF object.

DESCRIPTION:

Release resources associated with an Array sparse matrix multiplication. After this call `routehandle` becomes invalid.

routehandle Handle to the precomputed Route.

[noGarbage] If set to `.TRUE.` the object will be fully destroyed and removed from the ESMF garbage collection system. Note however that under this condition ESMF cannot protect against accessing the destroyed object through dangling aliases – a situation which may lead to hard to debug application crashes.

It is generally recommended to leave the `noGarbage` argument set to `.FALSE.` (the default), and to take advantage of the ESMF garbage collection system which will prevent problems with dangling aliases or incorrect sequences of destroy calls. However this level of support requires that a small remnant of the object is kept in memory past the destroy call. This can lead to an unexpected increase in memory consumption over the course of execution in applications that use temporary ESMF objects. For situations where the repeated creation and destruction of temporary objects leads to memory issues, it is recommended to call with `noGarbage` set to `.TRUE.`, fully removing the entire temporary object from memory.

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

28.5.39 ESMF_ArraySMMStore - Precompute Array sparse matrix multiplication with local factors

INTERFACE:

```
! Private name; call using ESMF_ArraySMMStore()
subroutine ESMF_ArraySMMStore<type><kind>(srcArray, dstArray, &
  routehandle, factorList, factorIndexList, &
  ignoreUnmatchedIndices, srcTermProcessing, pipelineDepth, rc)
```

ARGUMENTS:

```
  type(ESMF_Array),          intent(in)              :: srcArray
  type(ESMF_Array),          intent(inout)           :: dstArray
  type(ESMF_RouteHandle),    intent(inout)           :: routehandle
  <type>(ESMF_KIND_<kind>), target, intent(in)       :: factorList(:)
  integer(ESMF_KIND_<kind>), intent(in)              :: factorIndexList(:,:)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  logical,                   intent(in),    optional :: ignoreUnmatchedIndices
  integer,                   intent(inout), optional :: srcTermProcessing
  integer,                   intent(inout), optional :: pipelineDepth
  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:

  6.1.0 Added argument `srcTermProcessing`. Added argument `pipelineDepth`. The new arguments provide access to the tuning parameters affecting the sparse matrix execution.

  7.0.0 Added argument `transposeRoutehandle` to allow a handle to the transposed matrix operation to be returned.
  Added argument `ignoreUnmatchedIndices` to support sparse matrices that contain elements with indices that do not have a match within the source or destination Array.

  7.1.0r Removed argument `transposeRoutehandle` and provide it via interface overloading instead. This allows argument `srcArray` to stay strictly intent(in) for this entry point.

DESCRIPTION:

`ESMF_ArraySMMStore()` is a collective method across all PETs of the current Component. The interface of the method is overloaded, allowing – in principle – each PET to call into `ESMF_ArraySMMStore()` through a different entry point. Restrictions apply as to which combinations are sensible. All other combinations result in ESMF run time errors. The complete semantics of the `ESMF_ArraySMMStore()` method, as provided through the separate entry points shown in 28.5.39 and 28.5.41, is described in the following paragraphs as a whole.

Store an Array sparse matrix multiplication operation from `srcArray` to `dstArray`. PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* `factorList` and `factorIndexList` arguments.

Both `srcArray` and `dstArray` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 28.2.18 for details on the definition of *sequence indices*.

Source and destination Arrays, as well as the supplied `factorList` argument, may be of different <type><kind>. Further source and destination Arrays may differ in shape and number of elements.

It is erroneous to specify the identical Array object for `srcArray` and `dstArray` arguments.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_ArraySMM()` on any pair of Arrays that matches `srcArray` and `dstArray` in *type*, *kind*, and memory layout of the *distributed* dimensions. However, the size, number, and index order of *undistributed* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This method is overloaded for:
`ESMF_TYPEKIND_I4`, `ESMF_TYPEKIND_I8`,
`ESMF_TYPEKIND_R4`, `ESMF_TYPEKIND_R8`.

This call is *collective* across the current VM.

srcArray `ESMF_Array` with source data.

dstArray `ESMF_Array` with destination data. The data in this Array may be destroyed by this call.

routehandle Handle to the precomputed Route.

factorList List of non-zero coefficients.

factorIndexList Pairs of sequence indices for the factors stored in `factorList`.

The second dimension of `factorIndexList` steps through the list of pairs, i.e. `size(factorIndexList,2) == size(factorList)`. The first dimension of `factorIndexList` is either of size 2 or size 4.

In the *size 2 format* `factorIndexList(1,:)` specifies the sequence index of the source element in the `srcArray` while `factorIndexList(2,:)` specifies the sequence index of the destination element in `dstArray`. For this format to be a valid option source and destination Arrays must have matching number of tensor elements (the product of the sizes of all Array tensor dimensions). Under this condition an identity matrix can be applied within the space of tensor elements for each sparse matrix factor.

The *size 4 format* is more general and does not require a matching tensor element count. Here the `factorIndexList(1,:)` specifies the sequence index while `factorIndexList(2,:)` specifies the tensor sequence index of the source element in the `srcArray`. Further `factorIndexList(3,:)` specifies the sequence index and `factorIndexList(4,:)` specifies the tensor sequence index of the destination element in the `dstArray`.

See section 28.2.18 for details on the definition of Array *sequence indices* and *tensor sequence indices*.

[ignoreUnmatchedIndices] A logical flag that affects the behavior for when sequence indices in the sparse matrix are encountered that do not have a match on the `srcArray` or `dstArray` side. The default setting is `.false.`, indicating that it is an error when such a situation is encountered. Setting `ignoreUnmatchedIndices` to `.true.` ignores entries with unmatched indices.

[srcTermProcessing] The `srcTermProcessing` parameter controls how many source terms, located on the same PET and summing into the same destination element, are summed into partial sums on the source PET before being transferred to the destination PET. A value of 0 indicates that the entire arithmetic is done on the destination PET; source elements are neither multiplied by their factors nor added into partial sums before being sent off by the source PET. A value of 1 indicates that source elements are multiplied by their factors on the source side before being sent to the destination PET. Larger values of `srcTermProcessing` indicate the maximum number of terms in the partial sums on the source side.

Note that partial sums may lead to bit-for-bit differences in the results. See section **??** for an in-depth discussion of *all* bit-for-bit reproducibility aspects related to route-based communication methods.

The `ESMF_ArraySMMStore()` method implements an auto-tuning scheme for the `srcTermProcessing` parameter. The intent on the `srcTermProcessing` argument is `"inout"` in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `srcTermProcessing` parameter, and the auto-tuning phase is skipped. In this case the `srcTermProcessing` argument is not modified on return. If the provided argument is < 0, the `srcTermProcessing` parameter is determined internally using the auto-tuning scheme. In this case the `srcTermProcessing` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `srcTermProcessing` argument is omitted.

[pipelineDepth] The `pipelineDepth` parameter controls how many messages a PET may have outstanding during a sparse matrix exchange. Larger values of `pipelineDepth` typically lead to better performance. However, on some systems too large a value may lead to performance degradation, or runtime errors.

Note that the pipeline depth has no effect on the bit-for-bit reproducibility of the results. However, it may affect the performance reproducibility of the exchange.

The `ESMF_ArraySMMStore()` method implements an auto-tuning scheme for the `pipelineDepth` parameter. The intent on the `pipelineDepth` argument is `"inout"` in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `pipelineDepth` parameter, and the auto-tuning phase is skipped. In this case the `pipelineDepth` argument is not modified on return. If the provided argument is < 0, the `pipelineDepth` parameter is determined internally using the auto-tuning scheme. In this case the `pipelineDepth` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `pipelineDepth` argument is omitted.

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

28.5.40 ESMF_ArraySMMStore - Precompute Array sparse matrix multiplication and transpose with local factors

INTERFACE:

```
! Private name; call using ESMF_ArraySMMStore()
subroutine ESMF_ArraySMMStore<type><kind>TP(srcArray, dstArray, &
  routehandle, transposeRoutehandle, factorList, factorIndexList, &
  ignoreUnmatchedIndices, srcTermProcessing, pipelineDepth, &
  rc)
```

ARGUMENTS:

```
  type(ESMF_Array),          intent(inout)           :: srcArray
  type(ESMF_Array),          intent(inout)           :: dstArray
  type(ESMF_RouteHandle),    intent(inout)           :: routehandle
  type(ESMF_RouteHandle),    intent(inout)           :: transposeRoutehandle
  <type>(ESMF_KIND_<kind>), target, intent(in)       :: factorList(:)
  integer(ESMF_KIND_<kind>), intent(in)              :: factorIndexList(:,:)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  logical,                   intent(in),    optional :: ignoreUnmatchedIndices
  integer,                   intent(inout), optional :: srcTermProcessing
  integer,                   intent(inout), optional :: pipelineDepth
  integer,                   intent(out),   optional :: rc
```

DESCRIPTION:

`ESMF_ArraySMMStore()` is a collective method across all PETs of the current Component. The interface of the method is overloaded, allowing – in principle – each PET to call into `ESMF_ArraySMMStore()` through a different entry point. Restrictions apply as to which combinations are sensible. All other combinations result in ESMF run time errors. The complete semantics of the `ESMF_ArraySMMStore()` method, as provided through the separate entry points shown in 28.5.40 and 28.5.42, is described in the following paragraphs as a whole.

Store an Array sparse matrix multiplication operation from `srcArray` to `dstArray`. PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* `factorList` and `factorIndexList` arguments.

Both `srcArray` and `dstArray` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 28.2.18 for details on the definition of *sequence indices*.

Source and destination Arrays, as well as the supplied `factorList` argument, may be of different <type><kind>. Further source and destination Arrays may differ in shape and number of elements.

It is erroneous to specify the identical Array object for `srcArray` and `dstArray` arguments.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_ArraySMM()` on any pair of Arrays that matches `srcArray` and `dstArray` in *type*, *kind*, and memory layout of the *distributed* dimensions. However, the size, number, and index order of *undistributed* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This method is overloaded for:

ESMF_TYPEKIND_I4, ESMF_TYPEKIND_I8,
ESMF_TYPEKIND_R4, ESMF_TYPEKIND_R8.

This call is *collective* across the current VM.

srcArray ESMF_Array with source data. The data in this Array may be destroyed by this call.

dstArray ESMF_Array with destination data. The data in this Array may be destroyed by this call.

routehandle Handle to the precomputed Route.

[transposeRoutehandle] Handle to the transposed matrix operation. The transposed operation goes from dstArray to srcArray.

factorList List of non-zero coefficients.

factorIndexList Pairs of sequence indices for the factors stored in factorList.

The second dimension of factorIndexList steps through the list of pairs, i.e. size(factorIndexList,2) == size(factorList). The first dimension of factorIndexList is either of size 2 or size 4.

In the *size 2 format* factorIndexList(1,:) specifies the sequence index of the source element in the srcArray while factorIndexList(2,:) specifies the sequence index of the destination element in dstArray. For this format to be a valid option source and destination Arrays must have matching number of tensor elements (the product of the sizes of all Array tensor dimensions). Under this condition an identity matrix can be applied within the space of tensor elements for each sparse matrix factor.

The *size 4 format* is more general and does not require a matching tensor element count. Here the factorIndexList(1,:) specifies the sequence index while factorIndexList(2,:) specifies the tensor sequence index of the source element in the srcArray. Further factorIndexList(3,:) specifies the sequence index and factorIndexList(4,:) specifies the tensor sequence index of the destination element in the dstArray.

See section 28.2.18 for details on the definition of Array *sequence indices* and *tensor sequence indices*.

[ignoreUnmatchedIndices] A logical flag that affects the behavior for when sequence indices in the sparse matrix are encountered that do not have a match on the srcArray or dstArray side. The default setting is .false., indicating that it is an error when such a situation is encountered. Setting ignoreUnmatchedIndices to .true. ignores entries with unmatched indices.

[srcTermProcessing] The srcTermProcessing parameter controls how many source terms, located on the same PET and summing into the same destination element, are summed into partial sums on the source PET before being transferred to the destination PET. A value of 0 indicates that the entire arithmetic is done on the destination PET; source elements are neither multiplied by their factors nor added into partial sums before being sent off by the source PET. A value of 1 indicates that source elements are multiplied by their factors on the source side before being sent to the destination PET. Larger values of srcTermProcessing indicate the maximum number of terms in the partial sums on the source side.

Note that partial sums may lead to bit-for-bit differences in the results. See section **??** for an in-depth discussion of *all* bit-for-bit reproducibility aspects related to route-based communication methods.

The ESMF_ArraySMMStore() method implements an auto-tuning scheme for the srcTermProcessing parameter. The intent on the srcTermProcessing argument is "inout" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the srcTermProcessing parameter, and the auto-tuning phase is skipped. In this case the srcTermProcessing argument is not modified on return. If the provided argument is < 0, the srcTermProcessing parameter is determined internally using the auto-tuning scheme. In this case the srcTermProcessing argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional srcTermProcessing argument is omitted.

[pipelineDepth] The `pipelineDepth` parameter controls how many messages a PET may have outstanding during a sparse matrix exchange. Larger values of `pipelineDepth` typically lead to better performance. However, on some systems too large a value may lead to performance degradation, or runtime errors.

Note that the pipeline depth has no effect on the bit-for-bit reproducibility of the results. However, it may affect the performance reproducibility of the exchange.

The `ESMF_ArraySMMStore()` method implements an auto-tuning scheme for the `pipelineDepth` parameter. The intent on the `pipelineDepth` argument is "`inout`" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `pipelineDepth` parameter, and the auto-tuning phase is skipped. In this case the `pipelineDepth` argument is not modified on return. If the provided argument is < 0, the `pipelineDepth` parameter is determined internally using the auto-tuning scheme. In this case the `pipelineDepth` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `pipelineDepth` argument is omitted.

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

28.5.41 ESMF_ArraySMMStore - Precompute Array sparse matrix multiplication without local factors

INTERFACE:

```
! Private name; call using ESMF_ArraySMMStore()
subroutine ESMF_ArraySMMStoreNF(srcArray, dstArray, routehandle, &
  ignoreUnmatchedIndices, srcTermProcessing, pipelineDepth, &
  rc)
```

ARGUMENTS:

```
  type(ESMF_Array),       intent(in)              :: srcArray
  type(ESMF_Array),       intent(inout)           :: dstArray
  type(ESMF_RouteHandle), intent(inout)           :: routehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  logical,                intent(in),    optional :: ignoreUnmatchedIndices
  integer,                intent(inout), optional :: srcTermProcessing
  integer,                intent(inout), optional :: pipelineDepth
  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:

  6.1.0 Added argument `srcTermProcessing`. Added argument `pipelineDepth`. The new arguments provide access to the tuning parameters affecting the sparse matrix execution.

7.0.0 Added argument `transposeRoutehandle` to allow a handle to the transposed matrix operation to be returned.
Added argument `ignoreUnmatchedIndices` to support sparse matrices that contain elements with indices that do not have a match within the source or destination Array.

7.1.0r Removed argument `transposeRoutehandle` and provide it via interface overloading instead. This allows argument `srcArray` to stay strictly intent(in) for this entry point.

DESCRIPTION:

`ESMF_ArraySMMStore()` is a collective method across all PETs of the current Component. The interface of the method is overloaded, allowing – in principle – each PET to call into `ESMF_ArraySMMStore()` through a different entry point. Restrictions apply as to which combinations are sensible. All other combinations result in ESMF run time errors. The complete semantics of the `ESMF_ArraySMMStore()` method, as provided through the separate entry points shown in 28.5.39 and 28.5.41, is described in the following paragraphs as a whole.

Store an Array sparse matrix multiplication operation from `srcArray` to `dstArray`. PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* `factorList` and `factorIndexList` arguments.

Both `srcArray` and `dstArray` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 28.2.18 for details on the definition of *sequence indices*.

Source and destination Arrays, as well as the supplied `factorList` argument, may be of different <type><kind>. Further source and destination Arrays may differ in shape and number of elements.

It is erroneous to specify the identical Array object for `srcArray` and `dstArray` arguments.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_ArraySMM()` on any pair of Arrays that matches `srcArray` and `dstArray` in *type*, *kind*, and memory layout of the *distributed* dimensions. However, the size, number, and index order of *undistributed* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This call is *collective* across the current VM.

srcArray `ESMF_Array` with source data.

dstArray `ESMF_Array` with destination data. The data in this Array may be destroyed by this call.

routehandle Handle to the precomputed Route.

[ignoreUnmatchedIndices] A logical flag that affects the behavior for when sequence indices in the sparse matrix are encountered that do not have a match on the `srcArray` or `dstArray` side. The default setting is `.false.`, indicating that it is an error when such a situation is encountered. Setting `ignoreUnmatchedIndices` to `.true.` ignores entries with unmatched indices.

[srcTermProcessing] The `srcTermProcessing` parameter controls how many source terms, located on the same PET and summing into the same destination element, are summed into partial sums on the source PET before being transferred to the destination PET. A value of 0 indicates that the entire arithmetic is done on the destination PET; source elements are neither multiplied by their factors nor added into partial sums before being sent off by the source PET. A value of 1 indicates that source elements are multiplied by their factors on the source side before being sent to the destination PET. Larger values of `srcTermProcessing` indicate the maximum number of terms in the partial sums on the source side.

Note that partial sums may lead to bit-for-bit differences in the results. See section **??** for an in-depth discussion of *all* bit-for-bit reproducibility aspects related to route-based communication methods.

The ESMF_ArraySMMStore() method implements an auto-tuning scheme for the srcTermProcessing parameter. The intent on the srcTermProcessing argument is "inout" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the srcTermProcessing parameter, and the auto-tuning phase is skipped. In this case the srcTermProcessing argument is not modified on return. If the provided argument is < 0, the srcTermProcessing parameter is determined internally using the auto-tuning scheme. In this case the srcTermProcessing argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional srcTermProcessing argument is omitted.

[pipelineDepth] The pipelineDepth parameter controls how many messages a PET may have outstanding during a sparse matrix exchange. Larger values of pipelineDepth typically lead to better performance. However, on some systems too large a value may lead to performance degradation, or runtime errors.

Note that the pipeline depth has no effect on the bit-for-bit reproducibility of the results. However, it may affect the performance reproducibility of the exchange.

The ESMF_ArraySMMStore() method implements an auto-tuning scheme for the pipelineDepth parameter. The intent on the pipelineDepth argument is "inout" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the pipelineDepth parameter, and the auto-tuning phase is skipped. In this case the pipelineDepth argument is not modified on return. If the provided argument is < 0, the pipelineDepth parameter is determined internally using the auto-tuning scheme. In this case the pipelineDepth argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional pipelineDepth argument is omitted.

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

28.5.42 ESMF_ArraySMMStore - Precompute Array sparse matrix multiplication and transpose without local factors

INTERFACE:

```
! Private name; call using ESMF_ArraySMMStore()
subroutine ESMF_ArraySMMStoreNFTP(srcArray, dstArray, routehandle, &
  transposeRoutehandle, ignoreUnmatchedIndices, &
  srcTermProcessing, pipelineDepth, rc)
```

ARGUMENTS:

```
  type(ESMF_Array),       intent(inout)           :: srcArray
  type(ESMF_Array),       intent(inout)           :: dstArray
  type(ESMF_RouteHandle), intent(inout)           :: routehandle
  type(ESMF_RouteHandle), intent(inout)           :: transposeRoutehandle
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
  logical,                intent(in),    optional :: ignoreUnmatchedIndices
  integer,                intent(inout), optional :: srcTermProcessing
  integer,                intent(inout), optional :: pipelineDepth
  integer,                intent(out),   optional :: rc
```

DESCRIPTION:

ESMF_ArraySMMStore() is a collective method across all PETs of the current Component. The interface of the method is overloaded, allowing – in principle – each PET to call into ESMF_ArraySMMStore() through a different

entry point. Restrictions apply as to which combinations are sensible. All other combinations result in ESMF run time errors. The complete semantics of the `ESMF_ArraySMMStore()` method, as provided through the separate entry points shown in 28.5.40 and 28.5.42, is described in the following paragraphs as a whole.

Store an Array sparse matrix multiplication operation from `srcArray` to `dstArray`. PETs that specify non-zero matrix coefficients must use the <type><kind> overloaded interface and provide the `factorList` and `factorIndexList` arguments. Providing `factorList` and `factorIndexList` arguments with `size(factorList) = (/0/)` and `size(factorIndexList) = (/2,0/)` or `(/4,0/)` indicates that a PET does not provide matrix elements. Alternatively, PETs that do not provide matrix elements may also call into the overloaded interface *without* `factorList` and `factorIndexList` arguments.

Both `srcArray` and `dstArray` are interpreted as sequentialized vectors. The sequence is defined by the order of DistGrid dimensions and the order of tiles within the DistGrid or by user-supplied arbitrary sequence indices. See section 28.2.18 for details on the definition of *sequence indices*.

Source and destination Arrays, as well as the supplied `factorList` argument, may be of different <type><kind>. Further source and destination Arrays may differ in shape and number of elements.

It is erroneous to specify the identical Array object for `srcArray` and `dstArray` arguments.

The routine returns an `ESMF_RouteHandle` that can be used to call `ESMF_ArraySMM()` on any pair of Arrays that matches `srcArray` and `dstArray` in *type*, *kind*, and memory layout of the *distributed* dimensions. However, the size, number, and index order of *undistributed* dimensions may be different. See section **??** for a more detailed discussion of RouteHandle reusability.

This call is *collective* across the current VM.

srcArray `ESMF_Array` with source data. The data in this Array may be destroyed by this call.

dstArray `ESMF_Array` with destination data. The data in this Array may be destroyed by this call.

routehandle Handle to the precomputed Route.

[transposeRoutehandle] Handle to the transposed matrix operation. The transposed operation goes from `dstArray` to `srcArray`.

[ignoreUnmatchedIndices] A logical flag that affects the behavior for when sequence indices in the sparse matrix are encountered that do not have a match on the `srcArray` or `dstArray` side. The default setting is `.false.`, indicating that it is an error when such a situation is encountered. Setting `ignoreUnmatchedIndices` to `.true.` ignores entries with unmatched indices.

[srcTermProcessing] The `srcTermProcessing` parameter controls how many source terms, located on the same PET and summing into the same destination element, are summed into partial sums on the source PET before being transferred to the destination PET. A value of 0 indicates that the entire arithmetic is done on the destination PET; source elements are neither multiplied by their factors nor added into partial sums before being sent off by the source PET. A value of 1 indicates that source elements are multiplied by their factors on the source side before being sent to the destination PET. Larger values of `srcTermProcessing` indicate the maximum number of terms in the partial sums on the source side.

Note that partial sums may lead to bit-for-bit differences in the results. See section **??** for an in-depth discussion of *all* bit-for-bit reproducibility aspects related to route-based communication methods.

The `ESMF_ArraySMMStore()` method implements an auto-tuning scheme for the `srcTermProcessing` parameter. The intent on the `srcTermProcessing` argument is `"inout"` in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `srcTermProcessing` parameter, and the auto-tuning phase is skipped. In this case the `srcTermProcessing` argument is not modified on return. If the provided argument is < 0, the `srcTermProcessing` parameter is determined internally using the auto-tuning scheme. In this case the `srcTermProcessing` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `srcTermProcessing` argument is omitted.

[pipelineDepth] The `pipelineDepth` parameter controls how many messages a PET may have outstanding during a sparse matrix exchange. Larger values of `pipelineDepth` typically lead to better performance. However, on some systems too large a value may lead to performance degradation, or runtime errors.

Note that the pipeline depth has no effect on the bit-for-bit reproducibility of the results. However, it may affect the performance reproducibility of the exchange.

The `ESMF_ArraySMMStore()` method implements an auto-tuning scheme for the `pipelineDepth` parameter. The intent on the `pipelineDepth` argument is "`inout`" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `pipelineDepth` parameter, and the auto-tuning phase is skipped. In this case the `pipelineDepth` argument is not modified on return. If the provided argument is < 0, the `pipelineDepth` parameter is determined internally using the auto-tuning scheme. In this case the `pipelineDepth` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `pipelineDepth` argument is omitted.

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

28.5.43 ESMF_ArraySMMStore - Precompute sparse matrix multiplication using factors read from file.

INTERFACE:

```
   ! Private name; call using ESMF_ArraySMMStore()
   subroutine ESMF_ArraySMMStoreFromFile(srcArray, dstArray, filename, &
     routehandle, ignoreUnmatchedIndices, &
     srcTermProcessing, pipelineDepth, rc)

   ! ARGUMENTS:
     type(ESMF_Array),       intent(in)              :: srcArray
     type(ESMF_Array),       intent(inout)           :: dstArray
     character(len=*),       intent(in)              :: filename
     type(ESMF_RouteHandle), intent(inout)           :: routehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),    optional :: ignoreUnmatchedIndices
     integer,                intent(inout), optional :: srcTermProcessing
     integer,                intent(inout), optional :: pipeLineDepth
     integer,                intent(out),   optional :: rc
```

DESCRIPTION:

Compute an `ESMF_RouteHandle` using factors read from file.

The arguments are:

srcArray `ESMF_Array` with source data.

dstArray `ESMF_Array` with destination data. The data in this Array may be destroyed by this call.

filename Path to the file containing weights for creating an `ESMF_RouteHandle`. See (12.9) for a description of the SCRIP weight file format. Only "row", "col", and "S" variables are required. They must be one-dimensionsal with dimension "n_s".

routehandle Handle to the `ESMF_RouteHandle`.

[ignoreUnmatchedIndices] A logical flag that affects the behavior for when sequence indices in the sparse matrix are encountered that do not have a match on the `srcArray` or `dstArray` side. The default setting is `.false.`, indicating that it is an error when such a situation is encountered. Setting `ignoreUnmatchedIndices` to `.true.` ignores entries with unmatched indices.

[srcTermProcessing] The `srcTermProcessing` parameter controls how many source terms, located on the same PET and summing into the same destination element, are summed into partial sums on the source PET before being transferred to the destination PET. A value of 0 indicates that the entire arithmetic is done on the destination PET; source elements are neither multiplied by their factors nor added into partial sums before being sent off by the source PET. A value of 1 indicates that source elements are multiplied by their factors on the source side before being sent to the destination PET. Larger values of `srcTermProcessing` indicate the maximum number of terms in the partial sums on the source side.

Note that partial sums may lead to bit-for-bit differences in the results. See section **??** for an in-depth discussion of *all* bit-for-bit reproducibility aspects related to route-based communication methods.

The `ESMF_ArraySMMStore()` method implements an auto-tuning scheme for the `srcTermProcessing` parameter. The intent on the `srcTermProcessing` argument is "`inout`" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `srcTermProcessing` parameter, and the auto-tuning phase is skipped. In this case the `srcTermProcessing` argument is not modified on return. If the provided argument is < 0, the `srcTermProcessing` parameter is determined internally using the auto-tuning scheme. In this case the `srcTermProcessing` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `srcTermProcessing` argument is omitted.

[pipelineDepth] The `pipelineDepth` parameter controls how many messages a PET may have outstanding during a sparse matrix exchange. Larger values of `pipelineDepth` typically lead to better performance. However, on some systems too large a value may lead to performance degradation, or runtime errors.

Note that the pipeline depth has no effect on the bit-for-bit reproducibility of the results. However, it may affect the performance reproducibility of the exchange. The `ESMF_ArraySMMStore()` method implements an auto-tuning scheme for the `pipelineDepth` parameter. The intent on the `pipelineDepth` argument is "`inout`" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `pipelineDepth` parameter, and the auto-tuning phase is skipped. In this case the `pipelineDepth` argument is not modified on return. If the provided argument is < 0, the `pipelineDepth` parameter is determined internally using the auto-tuning scheme. In this case the `pipelineDepth` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `pipelineDepth` argument is omitted.

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

28.5.44 ESMF_ArraySMMStore - Precompute sparse matrix multiplication and transpose using factors read from file.

INTERFACE:

```
! Private name; call using ESMF_ArraySMMStore()
subroutine ESMF_ArraySMMStoreFromFileTP(srcArray, dstArray, filename, &
  routehandle, transposeRoutehandle, ignoreUnmatchedIndices,&
  srcTermProcessing, pipelineDepth, rc)

! ARGUMENTS:
  type(ESMF_Array),       intent(inout)           :: srcArray
```

```
    type(ESMF_Array),       intent(inout)           :: dstArray
    character(len=*),       intent(in)              :: filename
    type(ESMF_RouteHandle), intent(inout)           :: routehandle
    type(ESMF_RouteHandle), intent(inout)           :: transposeRoutehandle
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
    logical,                intent(in),    optional :: ignoreUnmatchedIndices
    integer,                intent(inout), optional :: srcTermProcessing
    integer,                intent(inout), optional :: pipeLineDepth
    integer,                intent(out),   optional :: rc
```

DESCRIPTION:

Compute an `ESMF_RouteHandle` using factors read from file.

The arguments are:

srcArray `ESMF_Array` with source data. The data in this Array may be destroyed by this call.

dstArray `ESMF_Array` with destination data. The data in this Array may be destroyed by this call.

filename Path to the file containing weights for creating an `ESMF_RouteHandle`. See (12.9) for a description of the SCRIP weight file format. Only "row", "col", and "S" variables are required. They must be one-dimensionsal with dimension "n_s".

routehandle Handle to the `ESMF_RouteHandle`.

[transposeRoutehandle] Handle to the transposed matrix operation. The transposed operation goes from `dstArray` to `srcArray`.

[ignoreUnmatchedIndices] A logical flag that affects the behavior for when sequence indices in the sparse matrix are encountered that do not have a match on the `srcArray` or `dstArray` side. The default setting is `.false.`, indicating that it is an error when such a situation is encountered. Setting `ignoreUnmatchedIndices` to `.true.` ignores entries with unmatched indices.

[srcTermProcessing] The `srcTermProcessing` parameter controls how many source terms, located on the same PET and summing into the same destination element, are summed into partial sums on the source PET before being transferred to the destination PET. A value of 0 indicates that the entire arithmetic is done on the destination PET; source elements are neither multiplied by their factors nor added into partial sums before being sent off by the source PET. A value of 1 indicates that source elements are multiplied by their factors on the source side before being sent to the destination PET. Larger values of `srcTermProcessing` indicate the maximum number of terms in the partial sums on the source side.

Note that partial sums may lead to bit-for-bit differences in the results. See section **??** for an in-depth discussion of *all* bit-for-bit reproducibility aspects related to route-based communication methods.

The `ESMF_ArraySMMStore()` method implements an auto-tuning scheme for the `srcTermProcessing` parameter. The intent on the `srcTermProcessing` argument is `"inout"` in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `srcTermProcessing` parameter, and the auto-tuning phase is skipped. In this case the `srcTermProcessing` argument is not modified on return. If the provided argument is < 0, the `srcTermProcessing` parameter is determined internally using the auto-tuning scheme. In this case the `srcTermProcessing` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `srcTermProcessing` argument is omitted.

[pipelineDepth] The `pipelineDepth` parameter controls how many messages a PET may have outstanding during a sparse matrix exchange. Larger values of `pipelineDepth` typically lead to better performance. However, on some systems too large a value may lead to performance degradation, or runtime errors.

Note that the pipeline depth has no effect on the bit-for-bit reproducibility of the results. However, it may affect the performance reproducibility of the exchange. The `ESMF_ArraySMMStore()` method implements an auto-tuning scheme for the `pipelineDepth` parameter. The intent on the `pipelineDepth` argument is "`inout`" in order to support both overriding and accessing the auto-tuning parameter. If an argument $>= 0$ is specified, it is used for the `pipelineDepth` parameter, and the auto-tuning phase is skipped. In this case the `pipelineDepth` argument is not modified on return. If the provided argument is < 0, the `pipelineDepth` parameter is determined internally using the auto-tuning scheme. In this case the `pipelineDepth` argument is re-set to the internally determined value on return. Auto-tuning is also used if the optional `pipelineDepth` argument is omitted.

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

28.5.45 ESMF_ArraySync - Synchronize DEs across the Array in case of sharing

INTERFACE:

```
   subroutine ESMF_ArraySync(array, rc)
```

ARGUMENTS:

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

DESCRIPTION:

Synchronizes access to DEs across `array` to make sure PETs correctly access the data for read and write when DEs are shared.

The arguments are:

array Specified `ESMF_Array` object.

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

28.5.46 ESMF_ArrayValidate - Validate object-wide Array information

INTERFACE:

```
   subroutine ESMF_ArrayValidate(array, rc)
```

ARGUMENTS:

```
     type(ESMF_Array), intent(in)            :: array
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     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:

Validates that the `Array` is internally consistent. The method returns an error code if problems are found.

The arguments are:

array Specified `ESMF_Array` object.

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

28.5.47 ESMF_ArrayWrite - Write Array data into a file

INTERFACE:

```
   subroutine ESMF_ArrayWrite(array, fileName, &
       variableName, convention, purpose,  &
       overwrite, status, timeslice, iofmt, rc)
```

ARGUMENTS:

```
     type(ESMF_Array),           intent(in)            :: array
     character(*),               intent(in)            :: fileName
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     character(*),               intent(in),  optional :: variableName
     character(*),               intent(in),  optional :: convention
     character(*),               intent(in),  optional :: purpose
     logical,                    intent(in),  optional :: overwrite
     type(ESMF_FileStatus_Flag), intent(in),  optional :: status
     integer,                    intent(in),  optional :: timeslice
     type(ESMF_IOFmt_Flag),      intent(in),  optional :: iofmt
     integer,                    intent(out), optional :: rc
```

DESCRIPTION:

Write Array data into a file. For this API to be functional, the environment variable `ESMF_PIO` should be set to "internal" when the ESMF library is built. Please see the section on Data I/O, **??**.

When `convention` and `purpose` arguments are specified, a NetCDF variable can be created with user-specified dimension labels and attributes. Dimension labels may be defined for both gridded and ungridded dimensions. Dimension labels for gridded dimensions are specified at the DistGrid level by attaching an ESMF Attribute package to it. The Attribute package must contain an attribute named by the pre-defined ESMF parameter `ESMF_ATT_GRIDDED_DIM_LABELS`. The corresponding value is an array of character strings specifying the desired names of the dimensions. Likewise, for ungridded dimensions, an Attribute package is attached at the Array level. The name of the name must be `ESMF_ATT_UNGRIDDED_DIM_LABELS`.

NetCDF attributes for the variable can also be specified. As with dimension labels, an Attribute package is added to the Array with the desired names and values. A value may be either a scalar character string, or a scalar or array of type integer, real, or double precision. Dimension label attributes can co-exist with variable attributes within a common Attribute package.

Limitations:

- Only single tile Arrays are supported.
- Not supported in `ESMF_COMM=mpiuni` mode.

The arguments are:

array The `ESMF_Array` object that contains data to be written.

fileName The name of the output file to which Array data is written.

[variableName] Variable name in the output file; default is the "name" of Array. Use this argument only in the I/O format (such as NetCDF) that supports variable name. If the I/O format does not support this (such as binary format), ESMF will return an error code.

[convention] Specifies an Attribute package associated with the Array, used to create NetCDF dimension labels and attributes for the variable in the file. When this argument is present, the `purpose` argument must also be present. Use this argument only with a NetCDF I/O format. If binary format is used, ESMF will return an error code.

[purpose] Specifies an Attribute package associated with the Array, used to create NetCDF dimension labels and attributes for the variable in the file. When this argument is present, the `convention` argument must also be present. Use this argument only with a NetCDF I/O format. If binary format is used, ESMF will return an error code.

[overwrite] A logical flag, the default is .false., i.e., existing Array data may *not* be overwritten. If .true., the overwrite behavior depends on the value of `iofmt` as shown below:

`iofmt` **=** `ESMF_IOFMT_BIN`**:** All data in the file will be overwritten with each Array's data.

`iofmt` **=** `ESMF_IOFMT_NETCDF, ESMF_IOFMT_NETCDF_64BIT_OFFSET`**:** Only the data corresponding to each Array's name will be be overwritten. If the `timeslice` option is given, only data for the given timeslice may be overwritten. Note that it is always an error to attempt to overwrite a NetCDF variable with data which has a different shape.

[status] The file status. Please see Section **??** for the list of options. If not present, defaults to `ESMF_FILESTATUS_UNKNOWN`.

[timeslice] Some I/O formats (e.g. NetCDF) support the output of data in form of time slices. An unlimited dimension called `time` is defined in the file variable for this capability. The `timeslice` argument provides access to the `time` dimension, and must have a positive value. The behavior of this option may depend on the setting of the `overwrite` flag:

`overwrite = .false.`**:** If the timeslice value is less than the maximum time already in the file, the write will fail.

`overwrite = .true.`**:** Any positive timeslice value is valid.

By default, i.e. by omitting the `timeslice` argument, no provisions for time slicing are made in the output file, however, if the file already contains a time axis for the variable, a timeslice one greater than the maximum will be written.

[iofmt] The I/O format. Please see Section **??** for the list of options. If not present, file names with a `.bin` extension will use `ESMF_IOFMT_BIN`, and file names with a `.nc` extension will use `ESMF_IOFMT_NETCDF`. Other files default to `ESMF_IOFMT_NETCDF`.

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

28.5.48 ESMF_SparseMatrixWrite - Write a sparse matrix to file

INTERFACE:

```
subroutine ESMF_SparseMatrixWrite(factorList, factorIndexList, fileName, &
  rc)
```

ARGUMENTS:

```
     real(ESMF_KIND_R8),    intent(in)            :: factorList(:)
     integer(ESMF_KIND_I4), intent(in)            :: factorIndexList(:,:)
     character(*),          intent(in)            :: fileName
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     integer,               intent(out), optional :: rc
```

DESCRIPTION:

Write the `factorList` and `factorIndexList` into a NetCDF file. The data is stored in SCRIP format documented under section (12.9).

Limitations:

- Only `real(ESMF_KIND_R8) factorList` and `integer(ESMF_KIND_I4) factorIndexList` supported.
- Not supported in `ESMF_COMM=mpiuni` mode.

The arguments are:

factorList The sparse matrix factors to be written.

factorIndexList The sparse matrix sequence indices to be written.

fileName The name of the output file to be written.

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

28.6 Class API: DynamicMask Methods

28.6.1 ESMF_DynamicMaskSetR8R8R8 - Set DynamicMask for R8R8R8

INTERFACE:

```
subroutine ESMF_DynamicMaskSetR8R8R8(dynamicMask, dynamicMaskRoutine, &
  handleAllElements, dynamicSrcMaskValue, &
  dynamicDstMaskValue, rc)
```

ARGUMENTS:

```
     type(ESMF_DynamicMask), intent(out)           :: dynamicMask
     procedure(ESMF_DynamicMaskRoutineR8R8R8)      :: dynamicMaskRoutine
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),  optional :: handleAllElements
     real(ESMF_KIND_R8),     intent(in),  optional :: dynamicSrcMaskValue
     real(ESMF_KIND_R8),     intent(in),  optional :: dynamicDstMaskValue
     integer,                intent(out), optional :: rc
```

DESCRIPTION:

Set an `ESMF_DynamicMask` object suitable for destination element typekind `ESMF_TYPEKIND_R8`, factor typekind `ESMF_TYPEKIND_R8`, and source element typekind `ESMF_TYPEKIND_R8`.

All values in `dynamicMask` will be reset by this call.

See section **??** for a general discussion of dynamic masking.

The arguments are:

dynamicMask DynamicMask object.

dynamicMaskRoutine The routine responsible for handling dynamically masked source and destination elements. See section **??** for the precise definition of the `dynamicMaskRoutine` procedure interface. The routine is only called on PETs where *at least one* interpolation element is identified for special handling.

[handleAllElements] If set to `.true.`, all local elements, regardless of their dynamic masking status, are made available to `dynamicMaskRoutine` for handling. This option can be used to implement fully customized interpolations based on the information provided by ESMF. The default is `.false.`, meaning that only elements affected by dynamic masking will be handed to `dynamicMaskRoutine`.

[dynamicSrcMaskValue] The value for which a source element is considered dynamically masked. The default is to *not* consider any source elements as dynamically masked.

[dynamicDstMaskValue] The value for which a destination element is considered dynamically masked. The default is to *not* consider any destination elements as dynamically masked.

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

28.6.2 ESMF_DynamicMaskSetR8R8R8V - Set DynamicMask for R8R8R8 with vectorization

INTERFACE:

```
  subroutine ESMF_DynamicMaskSetR8R8R8V(dynamicMask, dynamicMaskRoutine, &
    handleAllElements, dynamicSrcMaskValue, &
    dynamicDstMaskValue, rc)
```

ARGUMENTS:

```
    type(ESMF_DynamicMask), intent(out)           :: dynamicMask
    procedure(ESMF_DynamicMaskRoutineR8R8R8V)     :: dynamicMaskRoutine
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    logical,                intent(in),  optional :: handleAllElements
    real(ESMF_KIND_R8),     intent(in),  optional :: dynamicSrcMaskValue
    real(ESMF_KIND_R8),     intent(in),  optional :: dynamicDstMaskValue
    integer,                intent(out), optional :: rc
```

DESCRIPTION:

Set an `ESMF_DynamicMask` object suitable for destination element typekind `ESMF_TYPEKIND_R8`, factor typekind `ESMF_TYPEKIND_R8`, and source element typekind `ESMF_TYPEKIND_R8`.

All values in `dynamicMask` will be reset by this call.

See section **??** for a general discussion of dynamic masking.

The arguments are:

dynamicMask DynamicMask object.

dynamicMaskRoutine The routine responsible for handling dynamically masked source and destination elements. See section **??** for the precise definition of the `dynamicMaskRoutine` procedure interface. The routine is only called on PETs where *at least one* interpolation element is identified for special handling.

[handleAllElements] If set to `.true.`, all local elements, regardless of their dynamic masking status, are made available to `dynamicMaskRoutine` for handling. This option can be used to implement fully customized interpolations based on the information provided by ESMF. The default is `.false.`, meaning that only elements affected by dynamic masking will be handed to `dynamicMaskRoutine`.

[dynamicSrcMaskValue] The value for which a source element is considered dynamically masked. The default is to *not* consider any source elements as dynamically masked.

[dynamicDstMaskValue] The value for which a destination element is considered dynamically masked. The default is to *not* consider any destination elements as dynamically masked.

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

28.6.3 ESMF_DynamicMaskSetR4R8R4 - Set DynamicMask for R4R8R4

INTERFACE:

```
  subroutine ESMF_DynamicMaskSetR4R8R4(dynamicMask, dynamicMaskRoutine, &
    handleAllElements, dynamicSrcMaskValue, &
    dynamicDstMaskValue, rc)
```

ARGUMENTS:

```
    type(ESMF_DynamicMask), intent(out)           :: dynamicMask
    procedure(ESMF_DynamicMaskRoutineR4R8R4)      :: dynamicMaskRoutine
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
```

```
    logical,                intent(in),  optional :: handleAllElements
    real(ESMF_KIND_R4),     intent(in),  optional :: dynamicSrcMaskValue
    real(ESMF_KIND_R4),     intent(in),  optional :: dynamicDstMaskValue
    integer,                intent(out), optional :: rc
```

DESCRIPTION:

Set an ESMF_DynamicMask object suitable for destination element typekind ESMF_TYPEKIND_R4, factor typekind ESMF_TYPEKIND_R8, and source element typekind ESMF_TYPEKIND_R4.

All values in dynamicMask will be reset by this call.

See section **??** for a general discussion of dynamic masking.

The arguments are:

dynamicMask DynamicMask object.

dynamicMaskRoutine The routine responsible for handling dynamically masked source and destination elements. See section **??** for the precise definition of the dynamicMaskRoutine procedure interface. The routine is only called on PETs where *at least one* interpolation element is identified for special handling.

[handleAllElements] If set to .true., all local elements, regardless of their dynamic masking status, are made available to dynamicMaskRoutine for handling. This option can be used to implement fully customized interpolations based on the information provided by ESMF. The default is .false., meaning that only elements affected by dynamic masking will be handed to dynamicMaskRoutine.

[dynamicSrcMaskValue] The value for which a source element is considered dynamically masked. The default is to *not* consider any source elements as dynamically masked.

[dynamicDstMaskValue] The value for which a destination element is considered dynamically masked. The default is to *not* consider any destination elements as dynamically masked.

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

28.6.4 ESMF_DynamicMaskSetR4R8R4V - Set DynamicMask for R4R8R4 with vectorization

INTERFACE:

```
  subroutine ESMF_DynamicMaskSetR4R8R4V(dynamicMask, dynamicMaskRoutine, &
    handleAllElements, dynamicSrcMaskValue, &
    dynamicDstMaskValue, rc)
```

ARGUMENTS:

```
    type(ESMF_DynamicMask), intent(out)           :: dynamicMask
    procedure(ESMF_DynamicMaskRoutineR4R8R4V)     :: dynamicMaskRoutine
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    logical,                intent(in),  optional :: handleAllElements
    real(ESMF_KIND_R4),     intent(in),  optional :: dynamicSrcMaskValue
    real(ESMF_KIND_R4),     intent(in),  optional :: dynamicDstMaskValue
    integer,                intent(out), optional :: rc
```

DESCRIPTION:

Set an ESMF_DynamicMask object suitable for destination element typekind ESMF_TYPEKIND_R4, factor typekind ESMF_TYPEKIND_R8, and source element typekind ESMF_TYPEKIND_R4.

All values in dynamicMask will be reset by this call.

See section **??** for a general discussion of dynamic masking.

The arguments are:

dynamicMask DynamicMask object.

dynamicMaskRoutine The routine responsible for handling dynamically masked source and destination elements. See section **??** for the precise definition of the dynamicMaskRoutine procedure interface. The routine is only called on PETs where *at least one* interpolation element is identified for special handling.

[handleAllElements] If set to .true., all local elements, regardless of their dynamic masking status, are made available to dynamicMaskRoutine for handling. This option can be used to implement fully customized interpolations based on the information provided by ESMF. The default is .false., meaning that only elements affected by dynamic masking will be handed to dynamicMaskRoutine.

[dynamicSrcMaskValue] The value for which a source element is considered dynamically masked. The default is to *not* consider any source elements as dynamically masked.

[dynamicDstMaskValue] The value for which a destination element is considered dynamically masked. The default is to *not* consider any destination elements as dynamically masked.

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

28.6.5 ESMF_DynamicMaskSetR4R4R4 - Set DynamicMask for R4R4R4

INTERFACE:

```
   subroutine ESMF_DynamicMaskSetR4R4R4(dynamicMask, dynamicMaskRoutine, &
     handleAllElements, dynamicSrcMaskValue, &
     dynamicDstMaskValue, rc)
```

ARGUMENTS:

```
     type(ESMF_DynamicMask), intent(out)           :: dynamicMask
     procedure(ESMF_DynamicMaskRoutineR4R4R4)      :: dynamicMaskRoutine
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),  optional :: handleAllElements
     real(ESMF_KIND_R4),     intent(in),  optional :: dynamicSrcMaskValue
     real(ESMF_KIND_R4),     intent(in),  optional :: dynamicDstMaskValue
     integer,                intent(out), optional :: rc
```

DESCRIPTION:

Set an ESMF_DynamicMask object suitable for destination element typekind ESMF_TYPEKIND_R4, factor typekind ESMF_TYPEKIND_R4, and source element typekind ESMF_TYPEKIND_R4.

All values in dynamicMask will be reset by this call.

See section **??** for a general discussion of dynamic masking.

The arguments are:

dynamicMask DynamicMask object.

dynamicMaskRoutine The routine responsible for handling dynamically masked source and destination elements. See section **??** for the precise definition of the dynamicMaskRoutine procedure interface. The routine is only called on PETs where *at least one* interpolation element is identified for special handling.

[handleAllElements] If set to .true., all local elements, regardless of their dynamic masking status, are made available to dynamicMaskRoutine for handling. This option can be used to implement fully customized interpolations based on the information provided by ESMF. The default is .false., meaning that only elements affected by dynamic masking will be handed to dynamicMaskRoutine.

[dynamicSrcMaskValue] The value for which a source element is considered dynamically masked. The default is to *not* consider any source elements as dynamically masked.

[dynamicDstMaskValue] The value for which a destination element is considered dynamically masked. The default is to *not* consider any destination elements as dynamically masked.

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

28.6.6 ESMF_DynamicMaskSetR4R4R4V - Set DynamicMask for R4R4R4 with vectorization

INTERFACE:

```
   subroutine ESMF_DynamicMaskSetR4R4R4V(dynamicMask, dynamicMaskRoutine, &
     handleAllElements, dynamicSrcMaskValue, &
     dynamicDstMaskValue, rc)
```

ARGUMENTS:

```
     type(ESMF_DynamicMask), intent(out)           :: dynamicMask
     procedure(ESMF_DynamicMaskRoutineR4R4R4V)     :: dynamicMaskRoutine
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     logical,                intent(in),  optional :: handleAllElements
     real(ESMF_KIND_R4),     intent(in),  optional :: dynamicSrcMaskValue
     real(ESMF_KIND_R4),     intent(in),  optional :: dynamicDstMaskValue
     integer,                intent(out), optional :: rc
```

DESCRIPTION:

Set an ESMF_DynamicMask object suitable for destination element typekind ESMF_TYPEKIND_R4, factor typekind ESMF_TYPEKIND_R4, and source element typekind ESMF_TYPEKIND_R4.

All values in dynamicMask will be reset by this call.

See section **??** for a general discussion of dynamic masking.

The arguments are:

dynamicMask DynamicMask object.

dynamicMaskRoutine The routine responsible for handling dynamically masked source and destination elements. See section **??** for the precise definition of the dynamicMaskRoutine procedure interface. The routine is only called on PETs where *at least one* interpolation element is identified for special handling.

[handleAllElements] If set to .true., all local elements, regardless of their dynamic masking status, are made available to dynamicMaskRoutine for handling. This option can be used to implement fully customized interpolations based on the information provided by ESMF. The default is .false., meaning that only elements affected by dynamic masking will be handed to dynamicMaskRoutine.

[dynamicSrcMaskValue] The value for which a source element is considered dynamically masked. The default is to *not* consider any source elements as dynamically masked.

[dynamicDstMaskValue] The value for which a destination element is considered dynamically masked. The default is to *not* consider any destination elements as dynamically masked.

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

29 LocalArray Class

29.1 Description

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

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

29.2 Restrictions and Future Work

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

29.3 Class API

29.3.1 ESMF_LocalArrayAssignment(=) - LocalArray assignment

INTERFACE:

```
interface assignment(=)
localarray1 = localarray2
```

ARGUMENTS:

```
type(ESMF_LocalArray) :: localarray1
type(ESMF_LocalArray) :: localarray2
```

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:

Assign localarray1 as an alias to the same ESMF LocalArray object in memory as localarray2. If localarray2 is invalid, then localarray1 will be equally invalid after the assignment.

The arguments are:

localarray1 The `ESMF_LocalArray` object on the left hand side of the assignment.

localarray2 The `ESMF_LocalArray` object on the right hand side of the assignment.

29.3.2 ESMF_LocalArrayOperator(==) - LocalArray equality operator

INTERFACE:

```
interface operator(==)
if (localarray1 == localarray2) then ... endif
OR
result = (localarray1 == localarray2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_LocalArray), intent(in) :: localarray1
type(ESMF_LocalArray), intent(in) :: localarray2
```

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:

Test whether localarray1 and localarray2 are valid aliases to the same ESMF LocalArray object in memory. For a more general comparison of two ESMF LocalArrays, going beyond the simple alias test, the ESMF_LocalArrayMatch() function (not yet implemented) must be used.

The arguments are:

localarray1 The `ESMF_LocalArray` object on the left hand side of the equality operation.

localarray2 The `ESMF_LocalArray` object on the right hand side of the equality operation.

29.3.3 ESMF_LocalArrayOperator(/=) - LocalArray not equal operator

INTERFACE:

```
interface operator(/=)
if (localarray1 /= localarray2) then ... endif
OR
result = (localarray1 /= localarray2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_LocalArray), intent(in) :: localarray1
type(ESMF_LocalArray), intent(in) :: localarray2
```

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:

Test whether localarray1 and localarray2 are *not* valid aliases to the same ESMF LocalArray object in memory. For a more general comparison of two ESMF LocalArrays, going beyond the simple alias test, the ESMF_LocalArrayMatch() function (not yet implemented) must be used.

The arguments are:

localarray1 The `ESMF_LocalArray` object on the left hand side of the non-equality operation.

localarray2 The `ESMF_LocalArray` object on the right hand side of the non-equality operation.

29.3.4 ESMF_LocalArrayCreate – Create a LocalArray by explicitly specifying typekind and rank arguments

INTERFACE:

```
! Private name; call using ESMF_LocalArrayCreate()
function ESMF_LocalArrayCreateByTKR(typekind, rank, totalCount, &
  totalLBound, totalUBound, rc)
```

RETURN VALUE:

```
type(ESMF_LocalArray) :: ESMF_LocalArrayCreateByTKR
```

ARGUMENTS:

```
type(ESMF_TypeKind_Flag), intent(in) :: typekind
integer, intent(in) :: rank
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
integer, intent(in), optional :: totalCount(:)
integer, intent(in), optional :: totalLBound(:)
integer, intent(in), optional :: totalUBound(:)
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:

Create a new `ESMF_LocalArray` and allocate data space, which remains uninitialized. The return value is a new LocalArray.

The arguments are:

typekind Array typekind. See section **??** for valid values.

rank Array rank (dimensionality, 1D, 2D, etc). Maximum allowed is 7D.

[totalCount] The number of items in each dimension of the array. This is a 1D integer array the same length as the rank. The `count` argument may be omitted if both `totalLBound` and `totalUBound` arguments are present.

[totalLBound] An integer array of length rank, with the lower index for each dimension.

[totalUBound] An integer array of length rank, with the upper index for each dimension.

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

29.3.5 ESMF_LocalArrayCreate – Create a LocalArray by specifying an ArraySpec

INTERFACE:

```
! Private name; call using ESMF_LocalArrayCreate()
function ESMF_LocalArrayCreateBySpec(arrayspec, totalCount, &
  totalLBound, totalUBound, rc)
```

RETURN VALUE:

```
type(ESMF_LocalArray) :: ESMF_LocalArrayCreateBySpec
```

ARGUMENTS:

```
type(ESMF_ArraySpec), intent(in) :: arrayspec
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
integer, intent(in), optional :: totalCount(:)
integer, intent(in), optional :: totalLBound(:)
integer, intent(in), optional :: totalUBound(:)
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:

Create a new `ESMF_LocalArray` and allocate data space, which remains uninitialized. The return value is a new LocalArray.

The arguments are:

arrayspec ArraySpec object specifying typekind and rank.

[totalCount] The number of items in each dimension of the array. This is a 1D integer array the same length as the rank. The `count` argument may be omitted if both `totalLBound` and `totalUBound` arguments are present.

[totalLBound] An integer array of length rank, with the lower index for each dimension.

[totalUBound] An integer array of length rank, with the upper index for each dimension.

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

29.3.6 ESMF_LocalArrayCreate – Create a LocalArray from pre-existing LocalArray

INTERFACE:

```
! Private name; call using ESMF_LocalArrayCreate()
function ESMF_LocalArrayCreateCopy(localarray, rc)
```

RETURN VALUE:

```
type(ESMF_LocalArray) :: ESMF_LocalArrayCreateCopy
```

ARGUMENTS:

```
type(ESMF_LocalArray), intent(in) :: localarray
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
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:

Perform a deep copy of an existing `ESMF_LocalArray` object. The return value is a new LocalArray.

The arguments are:

localarray Existing LocalArray to be copied.

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

29.3.7 ESMF_LocalArrayCreate - Create a LocalArray from a Fortran pointer (associated or unassociated)

INTERFACE:

```
! Private name; call using ESMF_LocalArrayCreate()
function ESMF_LocalArrCreateByPtr<rank><type><kind>(farrayPtr, &
datacopyflag, totalCount, totalLBound, totalUBound, rc)
```

RETURN VALUE:

```
type(ESMF_LocalArray) :: ESMF_LocalArrCreateByPtr<rank><type><kind>
```

ARGUMENTS:

```
<type> (ESMF_KIND_<kind>), pointer :: farrayPtr(<rank>)
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
```

```
integer, intent(in), optional :: totalCount(:)
integer, intent(in), optional :: totalLBound(:)
integer, intent(in), optional :: totalUBound(:)
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:

Creates an ESMF_LocalArray based on a Fortran array pointer. Two cases must be distinguished.

First, if farrayPtr is associated the optional datacopyflag argument may be used to indicate whether the associated data is to be copied or referenced. For associated farrayPtr the optional totalCount, totalLBound and totalUBound arguments need not be specified. However, all present arguments will be checked against farrayPtr for consistency.

Second, if farrayPtr is unassociated the optional argument datacopyflag must not be specified. However, in this case a complete set of totalCount and bounds information must be provided. Any combination of present totalCount totalLBound and totalUBound arguments that provides a complete specification is valid. All input information will be checked for consistency.

The arguments are:

farrayPtr A Fortran array pointer (associated or unassociated).

[datacopyflag] Indicate copy vs. reference behavior in case of associated farrayPtr. This argument must *not* be present for unassociated farrayPtr. Default to ESMF_DATACOPY_REFERENCE, makes the ESMF_LocalArray reference the associated data array. If set to ESMF_DATACOPY_VALUE this routine allocates new memory and copies the data from the pointer into the new LocalArray allocation.

[totalCount] The number of items in each dimension of the array. This is a 1D integer array the same length as the rank. The count argument may be omitted if both totalLBound and totalUBound arguments are present.

[totalLBound] An integer array of lower index values. Must be the same length as the rank.

[totalUBound] An integer array of upper index values. Must be the same length as the rank.

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

29.3.8 ESMF_LocalArrayDestroy - Release resources associated with a LocalArray

INTERFACE:

```
subroutine ESMF_LocalArrayDestroy(localarray, rc)
```

ARGUMENTS:

```
    type(ESMF_LocalArray), intent(inout) :: localarray
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    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:

Destroys an `ESMF_LocalArray`, releasing all resources associated with the object.

The arguments are:

localarray Destroy contents of this `ESMF_LocalArray`.

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

29.3.9 ESMF_LocalArrayGet - Get object-wide LocalArray information

INTERFACE:

```
  ! Private name; call using ESMF_LocalArrayGet()
  subroutine ESMF_LocalArrayGetDefault(localarray, &
    typekind, rank, totalCount, totalLBound, totalUBound, rc)
```

ARGUMENTS:

```
    type(ESMF_LocalArray), intent(in) :: localarray
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    type(ESMF_TypeKind_Flag), intent(out), optional :: typekind
    integer, intent(out), optional :: rank
    integer, intent(out), optional :: totalCount(:)
    integer, intent(out), optional :: totalLBound(:)
    integer, intent(out), optional :: totalUBound(:)
    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:

Returns information about the `ESMF_LocalArray`.

The arguments are:

localarray Queried `ESMF_LocalArray` object.

[typekind] TypeKind of the LocalArray object.

[rank] Rank of the LocalArray object.

[totalCount] Count per dimension.

[totalLBound] Lower bound per dimension.

[totalUBound] Upper bound per dimension.

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

29.3.10 ESMF_LocalArrayGet - Get a Fortran array pointer from a LocalArray

INTERFACE:

```
! Private name; call using ESMF_LocalArrayGet()
subroutine ESMF_LocalArrayGetData<rank><type><kind>(localarray, farrayPtr, &
datacopyflag, rc)
```

ARGUMENTS:

```
type(ESMF_LocalArray) :: localarray
<type> (ESMF_KIND_<kind>), pointer :: farrayPtr
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
type(ESMF_DataCopy_Flag), intent(in), optional :: datacopyflag
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:

Return a Fortran pointer to the data buffer, or return a Fortran pointer to a new copy of the data.

The arguments are:

localarray The `ESMF_LocalArray` to get the value from.

farrayPtr An unassociated or associated Fortran pointer correctly allocated.

[datacopyflag] An optional copy flag which can be specified. Can either make a new copy of the data or reference existing data. See section **??** for a list of possible values.

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

29.3.11 ESMF_LocalArrayIsCreated - Check whether a LocalArray object has been created

INTERFACE:

```
function ESMF_LocalArrayIsCreated(localarray, rc)
```

RETURN VALUE:

```
logical :: ESMF_LocalArrayIsCreated
```

ARGUMENTS:

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

DESCRIPTION:

Return `.true.` if the `localarray` has been created. Otherwise return `.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:

localarray `ESMF_LocalArray` queried.

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

---

30 ArraySpec Class

30.1 Description

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

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

30.2 Use and Examples

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

```
! !PROGRAM: ESMF_ArraySpecEx - ArraySpec manipulation examples
!
! !DESCRIPTION:
!
! This program shows examples of ArraySpec set and get usage
!-------------------------------------------------------------------------
#include "ESMF.h"

      ! ESMF Framework module
      use ESMF
      use ESMF_TestMod
      implicit none

      ! local variables
      type(ESMF_ArraySpec) :: arrayDS
      integer :: myrank
      type(ESMF_TypeKind_Flag) :: mytypekind


      ! return code
      integer:: rc, result
      character(ESMF_MAXSTR) :: testname
      character(ESMF_MAXSTR) :: failMsg


      ! initialize ESMF framework
      call ESMF_Initialize(defaultlogfilename="ArraySpecEx.Log", &
                    logkindflag=ESMF_LOGKIND_MULTI, rc=rc)
```

30.2.1 Set ArraySpec values

This example shows how to set values in an `ESMF_ArraySpec`.

```
      call ESMF_ArraySpecSet(arrayDS, rank=2, &
                             typekind=ESMF_TYPEKIND_R8, rc=rc)
```

30.2.2 Get ArraySpec values

This example shows how to query an `ESMF_ArraySpec`.

```
      call ESMF_ArraySpecGet(arrayDS, rank=myrank, &
        typekind=mytypekind, rc=rc)
      print *, "Returned values from ArraySpec:"
      print *, "rank =", myrank
```

```
! finalize ESMF framework
call ESMF_Finalize(rc=rc)


end program ESMF_ArraySpecEx
```

30.3 Restrictions and Future Work

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

30.4 Design and Implementation Notes

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

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

30.5 Class API

30.5.1 ESMF_ArraySpecAssignment(=) - Assign an ArraySpec to another ArraySpec

INTERFACE:

```
interface assignment(=)
  arrayspec1 = arrayspec2
```

ARGUMENTS:

```
type(ESMF_ArraySpec) :: arrayspec1
type(ESMF_ArraySpec) :: arrayspec2
```

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:

Set `arrayspec1` equal to `arrayspec2`. This is the default Fortran assignment, which creates a complete, independent copy of `arrayspec2` as `arrayspec1`. If `arrayspec2` is an invalid `ESMF_ArraySpec` object then `arrayspec1` will be equally invalid after the assignment.

The arguments are:

arrayspec1 The `ESMF_ArraySpec` to be set.

arrayspec2 The `ESMF_ArraySpec` to be copied.

30.5.2 ESMF_ArraySpecOperator(==) - Test if ArraySpec 1 is equal to ArraySpec 2

INTERFACE:

```
interface operator(==)
  if (arrayspec1 == arrayspec2) then ... endif
                OR
  result = (arrayspec1 == arrayspec2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_ArraySpec), intent(in) :: arrayspec1
type(ESMF_ArraySpec), intent(in) :: arrayspec2
```

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:

Overloads the (==) operator for the `ESMF_ArraySpec` class to return `.true.` if `arrayspec1` and `arrayspec2` specify the same type, kind and rank, and `.false.` otherwise.

The arguments are:

arrayspec1 First `ESMF_ArraySpec` in comparison.

arrayspec2 Second `ESMF_ArraySpec` in comparison.

30.5.3 ESMF_ArraySpecOperator(/=) - Test if ArraySpec 1 is not equal to ArraySpec 2

INTERFACE:

```
interface operator(/=)
  if (arrayspec1 /= arrayspec2) then ... endif
                OR
  result = (arrayspec1 /= arrayspec2)
```

RETURN VALUE:

```
logical :: result
```

ARGUMENTS:

```
type(ESMF_ArraySpec), intent(in) :: arrayspec1
type(ESMF_ArraySpec), intent(in) :: arrayspec2
```

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:

Overloads the (/=) operator for the `ESMF_ArraySpec` class to return `.true.` if `arrayspec1` and `arrayspec2` do not specify the same type, kind or rank, and `.false.` otherwise.

The arguments are:

arrayspec1 First `ESMF_ArraySpec` in comparison.

arrayspec2 Second `ESMF_ArraySpec` in comparison.

30.5.4 ESMF_ArraySpecGet - Get values from an ArraySpec

INTERFACE:

```
subroutine ESMF_ArraySpecGet(arrayspec, rank, typekind, rc)
```

ARGUMENTS:

```
 type(ESMF_ArraySpec),     intent(in)            :: arrayspec
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
 integer,                  intent(out), optional :: rank
 type(ESMF_TypeKind_Flag), intent(out), optional :: typekind
 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:

Returns information about the contents of an `ESMF_ArraySpec`.

The arguments are:

arrayspec The `ESMF_ArraySpec` to query.

[rank] Array rank (dimensionality – 1D, 2D, etc). Maximum possible is 7D.

[typekind] Array typekind. See section **??** for valid values.

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

30.5.5 ESMF_ArraySpecPrint - Print ArraySpec information

INTERFACE:

```
   subroutine ESMF_ArraySpecPrint(arrayspec, rc)
```

ARGUMENTS:

```
     type(ESMF_ArraySpec), intent(in)            :: arrayspec
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     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:

Print ArraySpec internals.

The arguments are:

arrayspec Specified `ESMF_ArraySpec` object.

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

30.5.6 ESMF_ArraySpecSet - Set values for an ArraySpec

INTERFACE:

```
   subroutine ESMF_ArraySpecSet(arrayspec, rank, typekind, rc)
```

ARGUMENTS:

```
     type(ESMF_ArraySpec),     intent(out)           :: arrayspec
     integer,                  intent(in)            :: rank
     type(ESMF_TypeKind_Flag), intent(in)            :: typekind
 -- The following arguments require argument keyword syntax (e.g. rc=rc). --
     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:

Creates a description of the data – the typekind, the rank, and the dimensionality.

The arguments are:

arrayspec The `ESMF_ArraySpec` to set.

rank Array rank (dimensionality – 1D, 2D, etc). Maximum allowed is 7D.

typekind Array typekind. See section **??** for valid values.

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

30.5.7 ESMF_ArraySpecValidate - Validate ArraySpec internals

INTERFACE:

```
subroutine ESMF_ArraySpecValidate(arrayspec, rc)
```

ARGUMENTS:

```
    type(ESMF_ArraySpec), intent(in)            :: arrayspec
-- The following arguments require argument keyword syntax (e.g. rc=rc). --
    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:

Validates that the `arrayspec` is internally consistent. The method returns an error code if problems are found.

The arguments are:

arrayspec Specified `ESMF_ArraySpec` object.

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

31 Grid Class

31.1 Description

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

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

31.1.1 Grid Representation in ESMF

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

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

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

Every ESMF Grid contains a DistGrid object, which defines the Grid's index space, topology, distribution, and connectivities. It enables the user to define the complex edge relationships of tripole and other grids. The DistGrid can be created explicitly and passed into a Grid creation routine, or it can be created implicitly if the user takes a Grid creation shortcut. The DistGrid used in Grid creation describes the properties of the Grid cells. In addition to this one, the Grid internally creates DistGrids for each stagger location. These stagger DistGrids are related to the original DistGrid, but may contain extra padding to represent the extent of the index space of the stagger. These DistGrids are what are used when a Field is created on a Grid.

31.1.2 Supported Grids

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

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

31.1.3 Grid Topologies and Periodicity

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

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

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

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

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

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

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

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

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

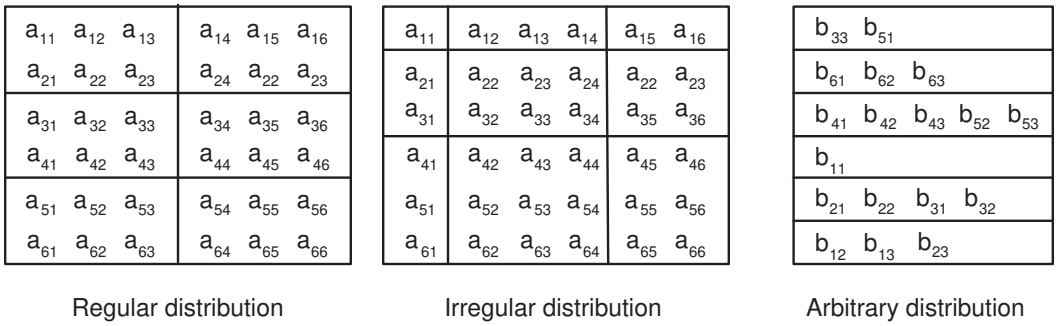


Figure 13: Examples of regular and irregular decomposition of a grid **a** that is 6x6, and an arbitrary decomposition of a grid **b** that is 6x3.

31.1.4 Grid Distribution

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

The main distribution options are regular, irregular, and arbitrary. A **regular** distribution is one in which the same number of contiguous grid cells are assigned to each DE in the distributed dimension. An **irregular** distribution is one in which unequal numbers of contiguous grid cells are assigned to each DE in the distributed dimension. An **arbitrary** distribution is one in which any grid cell can be assigned to any DE. Any of these distribution options can be applied to any of the grid shapes (i.e., rectangle) or types (i.e., rectilinear). Support for arbitrary distribution is limited in the current version of ESMF, see Section 31.3.7 for an example of creating a Grid with an arbitrary distribution.

Figure 13 illustrates options for distribution.

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

31.1.5 Grid Coordinates

Grid Tiles can have uniform, rectilinear, or curvilinear coordinates. The coordinates of **uniform** grids are equally spaced along their axes, and can be fully specified by the coordinates of the two opposing points that define the grid's physical span. The coordinates of **rectilinear** grids are unequally spaced along their axes, and can be fully specified by giving the spacing of grid points along each axis. The coordinates of **curvilinear grids** must be specified by giving the explicit set of coordinates for each grid point. Curvilinear grids are often uniform or rectilinear grids that have been warped; for example, to place a pole over a land mass so that it does not affect the computations performed on an ocean model grid. Figure 14 shows examples of each type of grid.

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

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

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

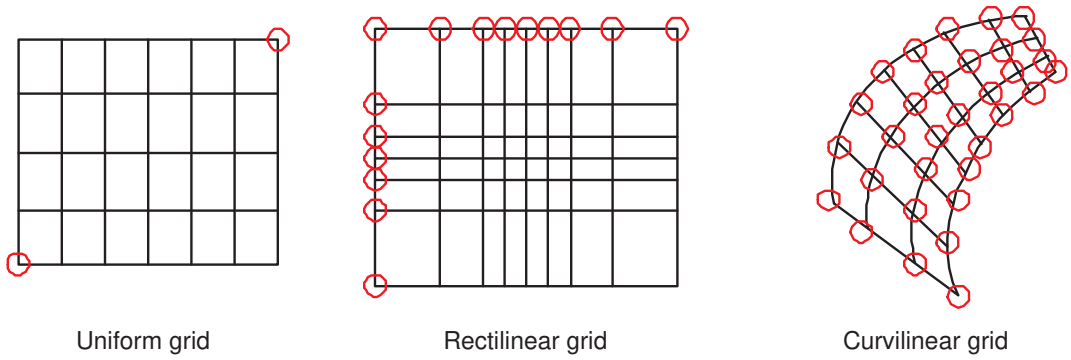


Figure 14: Types of logically rectangular grid tiles. Red circles show the values needed to specify grid coordinates for each type.

31.1.6 Coordinate Specification and Generation

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

See Section 31.3.13 for more description and examples of setting coordinates.

31.1.7 Staggering

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

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

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

In the current release, only the cell center stagger location is supported for an arbitrarily distributed grid. For examples and a full description of the stagger interface see Section 31.3.13.

31.1.8 Masking

Masking is the process whereby parts of a Grid can be marked to be ignored during an operation. For a description of how to set mask information in the Grid, see here 31.3.17. For a description of how masking works in regridding, see here 24.2.10.

31.2 Constants

31.2.1 ESMF_GRIDCONN

DESCRIPTION:

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

The type of this flag is:

`type(ESMF_GridConn_Flag)`

The valid values are:

ESMF_GRIDCONN_NONE No connection.

ESMF_GRIDCONN_PERIODIC Periodic connection.

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

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

31.2.2 ESMF_GRIDITEM

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

The type of this flag is:

`type(ESMF_GridItem_Flag)`

The valid values are:

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

NOTE: One important thing to consider when setting areas in the Grid using `ESMF_GRIDITEM_AREA`, ESMF doesn't currently do unit conversion on areas. If these areas are going to be used in a process that also involves the areas of another Grid or Mesh (e.g. conservative regridding), then it is the user's responsibility to make sure that the area units are consistent between the two sides. If ESMF calculates an area on the surface of a sphere, then it is in units of square radians. If it calculates the area for a Cartesian grid, then it is in the same units as the coordinates, but squared.

31.2.3 ESMF_GRIDMATCH

DESCRIPTION:
This type is used to indicate the level to which two grids match.

The type of this flag is:

`type(ESMF_GridMatch_Flag)`

The valid values are:

ESMF_GRIDMATCH_INVALID: Indicates a non-valid matching level. Returned if an error occurs in the matching function. If a higher matching level is returned then no error occurred.

ESMF_GRIDMATCH_NONE: The lowest level of grid matching. This indicates that the Grid's don't match at any of the higher levels.

ESMF_GRIDMATCH_EXACT: All the pieces of the Grid (e.g. distgrids, coordinates, etc.) except the name, match between the two Grids.

ESMF_GRIDMATCH_ALIAS: Both Grid variables are aliases to the exact same Grid object in memory.

31.2.4 ESMF_GRIDSTATUS

DESCRIPTION:
The ESMF Grid class can exist in two states. These states are present so that the library code can detect if a Grid has been appropriately setup for the task at hand. The following are the valid values of ESMF_GRIDSTATUS.

The type of this flag is:

`type(ESMF_GridStatus_Flag)`

The valid values are:

ESMF_GRIDSTATUS_EMPTY: Status after a Grid has been created with `ESMF_GridEmptyCreate`. A Grid object container is allocated but space for internal objects is not. Topology information and coordinate information is incomplete. This object can be used in `ESMF_GridEmptyComplete()` methods in which additional information is added to the Grid.

ESMF_GRIDSTATUS_COMPLETE: The Grid has a specific topology and distribution, but incomplete coordinate arrays. The Grid can be used as the basis for allocating a Field, and coordinates can be added via `ESMF_GridCoordAdd()` to allow other functionality.

31.2.5 ESMF_POLEKIND

DESCRIPTION:
This type describes the type of connection that occurs at the pole when a Grid is created with `ESMF_GridCreate1PeriodicDim()`.

The type of this flag is:

`type(ESMF_PoleKind_Flag)`

The valid values are:

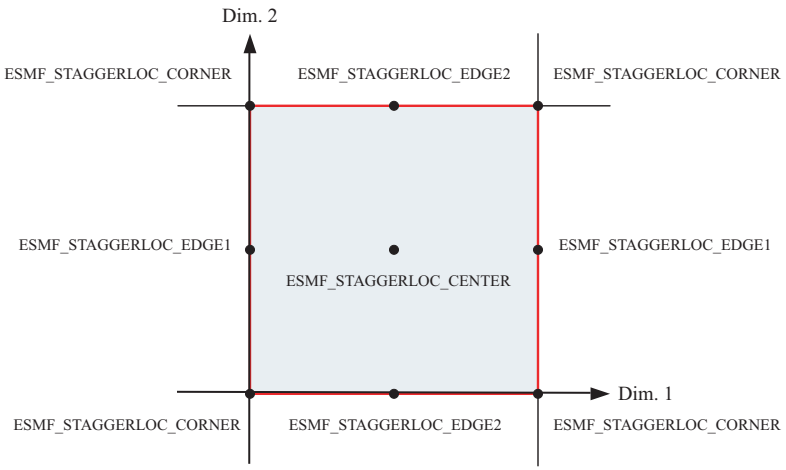


Figure 15: 2D Predefined Stagger Locations

ESMF_POLEKIND_NONE No connection at pole.

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

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

31.2.6 ESMF_STAGGERLOC

DESCRIPTION:
In the ESMF Grid class, data can be located at different positions in a Grid cell. When setting or retrieving coordinate data the stagger location is specified to tell the Grid method from where in the cell to get the data. Although the user may define their own custom stagger locations, ESMF provides a set of predefined locations for ease of use. The following are the valid predefined stagger locations.

The 2D predefined stagger locations (illustrated in figure 15) are:

ESMF_STAGGERLOC_CENTER: The center of the cell.

ESMF_STAGGERLOC_CORNER: The corners of the cell.

ESMF_STAGGERLOC_EDGE1: The edges offset from the center in the 1st dimension.

ESMF_STAGGERLOC_EDGE2: The edges offset from the center in the 2nd dimension.

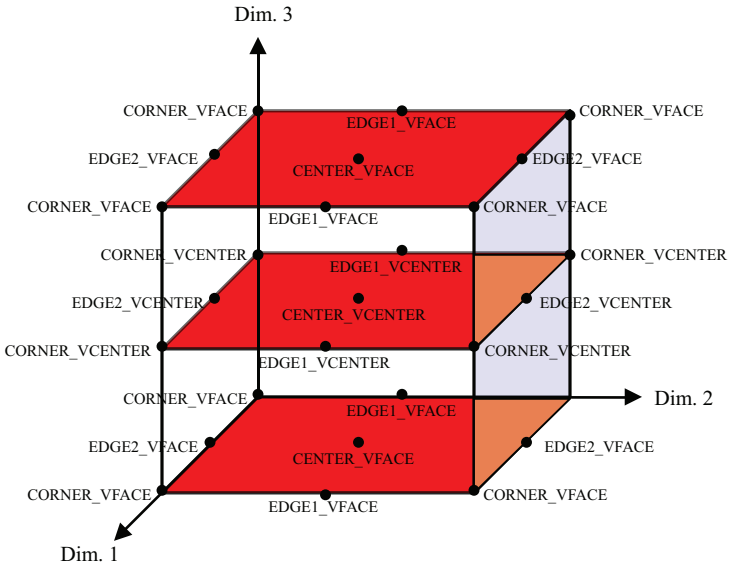


Figure 16: 3D Predefined Stagger Locations

The 3D predefined stagger locations (illustrated in figure 16) are:

ESMF_STAGGERLOC_CENTER_VCENTER: The center of the 3D cell.

ESMF_STAGGERLOC_CORNER_VCENTER: Half way up the vertical edges of the cell.

ESMF_STAGGERLOC_EDGE1_VCENTER: The center of the face bounded by edge 1 and the vertical dimension.

ESMF_STAGGERLOC_EDGE2_VCENTER: The center of the face bounded by edge 2 and the vertical dimension.

ESMF_STAGGERLOC_CORNER_VFACE: The corners of the 3D cell.

ESMF_STAGGERLOC_EDGE1_VFACE: The center of the edges of the 3D cell parallel offset from the center in the 1st dimension.

ESMF_STAGGERLOC_EDGE2_VFACE: The center of the edges of the 3D cell parallel offset from the center in the 2nd dimension.

ESMF_STAGGERLOC_CENTER_VFACE: The center of the top and bottom face. The face bounded by the 1st and 2nd dimensions.

31.3 Use and Examples

This section describes the use of the ESMF Grid class. It first discusses the more user friendly shape specific interface to the Grid. During this discussion it covers creation and options, adding stagger locations, coordinate data access,

and other grid functionality. After this initial phase the document discusses the more advanced options which the user can employ should they need more customized interaction with the Grid class.

31.3.1 Create single-tile Grid shortcut method

The set of methods `ESMF_GridCreateNoPeriDim()`, `ESMF_GridCreate1PeriDim()`, `ESMF_GridCreate2PeriDim()`, and `ESMF_GridCreate()` are shortcuts for building 2D or 3D single tile logically rectangular Grids. These methods support all three types of distributions described in Section 31.1.4: regular, irregular and arbitrary.

The ESMF Grid is cell based and so for all distribution options the methods take as input the number of cells to describe the total index space and the number of cells to specify distribution.

To create a Grid with a regular distribution the user specifies the global maximum and minimum ranges of the Grid cell index space (`maxIndex` and `minIndex`), and the number of pieces in which to partition each dimension (via a `regDecomp` argument). ESMF then divides the index space as evenly as possible into the specified number of pieces. If there are cells left over then they are distributed one per DE starting from the first DE until they are gone.

If `minIndex` is not specified, then the bottom of the Grid cell index range is assumed to be (1,1,...,1). If `regDecomp` is not specified, then by default ESMF creates a distribution that partitions the grid cells in the first dimension (e.g. NPx1x1...1) as evenly as possible by the number of PETs NP. The remaining dimensions are not partitioned. The dimension of the Grid is the size of `maxIndex`. The following is an example of creating a 10x20x30 3D grid where the first dimensions is broken into 2 pieces, the second is broken into 4 pieces, and the third is not divided (i.e. every DE will have length 30 in the 3rd dimension).

```
grid3D=ESMF_GridCreateNoPeriDim(regDecomp=(/2,4,1/), maxIndex=(/10,20,30/), &
          rc=rc)
```

Irregular distribution requires the user to specify the exact number of Grid cells per DE in each dimension. In the `ESMF_GridCreateNoPeriDim()` call the `countsPerDEDim1`, `countsPerDim2`, and `countsPerDim3` arguments are used to specify a rectangular distribution containing size(countsPerDEDim1) by size(countsPerDEDim2) by size(countsPerDEDim3) DEs. The entries in each of these arrays specify the number of grid cells per DE in that dimension. The dimension of the grid is determined by the presence of `countsPerDEDim3`. If it's present the Grid will be 3D. If just `countsPerDEDim1` and `countsPerDEDim2` are specified the Grid will be 2D.

The following call illustrates the creation of a 10x20 two dimensional rectangular Grid distributed across six DEs that are arranged 2x3. In the first dimension there are 3 grid cells on the first DE and 7 cells on the second DE. The second dimension has 3 DEs with 11,2, and 7 cells, respectively.

```
grid2D=ESMF_GridCreateNoPeriDim(countsPerDEDim1=(/3,7/), &
       countsPerDEDim2=(/11,2,7/), rc=rc)
```

To add a distributed third dimension of size 30, broken up into two groups of 15, the above call would be altered as follows.

```
grid3d=ESMF_GridCreateNoPeriDim(countsPerDEDim1=(/3,7/), &
       countsPerDEDim2=(/11,2,7/), countsPerDEDim3=(/15,15/), rc=rc)
```

To make a third dimension distributed across only 1 DE, then `countsPerDEDim3` in the call should only have a single term.

```
grid3D=ESMF_GridCreateNoPeriDim(countsPerDEDim1=(/3,7/),  &
       countsPerDEDim2=(/11,2,7/), countsPerDEDim3=(/30/), rc=rc)
```

The petMap parameter may be used to specify on to which specific PETs the DEs in the Grid are assigned. Each entry in petMap specifies to which PET the corresponding DE should be assigned. For example, petMap(3,2)=4 tells the Grid create call to put the DE located at column 3 row 2 on PET 4. Note that this parameter is only available for the regular and irregular distribution types. The petMap array is a 3D array, for a 3D Grid each of its dimensions correspond to a Grid dimension. If the Grid is 2D, then the first two dimensions correspond to Grid dimensions and the last dimension should be of size 1. The size of each petMap dimension is the number of DE's along that dimension in the Grid. For a regular Grid, the size is equal to the number in regDecomp (i.e. size(petMap,d)=regDecomp(d) for all dimensions d in the Grid). For an irregular Grid the size is equal to the number of items in the corresponding countsPerDEDim variable (i.e. size(petMap,d)=size(countsPerDEDimd) for all dimensions d in the Grid). The following example demonstrates how to specify the PET to DE association for an ESMF_GridCreateNoPeriDim() call.

```
! allocate memory for petMap
allocate( petMap(2,2,1) )

! Set petMap
petMap(:,1,1) = (/3,2/) ! DE (1,1,1) on PET 3 and DE (2,1,1) on PET 2
petMap(:,2,1) = (/1,0/) ! DE (1,2,1) on PET 1 and DE (2,2,1) on PET 0


! Let the 3D grid be be distributed only in the first two dimensions.
grid2D=ESMF_GridCreateNoPeriDim(countsPerDEDim1=(/3,7/), &
        countsPerDEDim2=(/7,6/), petMap=petMap, rc=rc)
```

To create an grid with arbitrary distribution, the user specifies the global minimum and maximum ranges of the index space with the arguments minIndex and maxIndex, the total number of cells and their index space locations residing on the local PET through a localArbIndexCount and a localArbIndex argument. localArbIndex is a 2D array with size (localArbIndexCount, n) where n is the total number dimensions distributed arbitrarily. Again, if minIndex is not specified, then the bottom of the index range is assumed to be (1,1,...). The dimension of the Grid is equal to the size of maxIndex. If n (number of arbitrarily distributed dimension) is less than the grid dimension, an optional argument distDim is used to specify which of the grid dimension is arbitrarily distributed. If not given, the first n dimensions are assumed to be distributed.

The following example creates a 2D Grid of dimensions 5x5, and places the diagonal elements (i.e. indices (i,i) where i goes from 1 to 5) on the local PET. The remaining PETs would individually declare the remainder of the Grid locations.

```
! allocate memory for localArbIndex
allocate( localArbIndex(5,2) )
! Set local indices
localArbIndex(1,:)=(/1,1/)
localArbIndex(2,:)=(/2,2/)
localArbIndex(3,:)=(/3,3/)
localArbIndex(4,:)=(/4,4/)
localArbIndex(5,:)=(/5,5/)

! Create a 2D Arbitrarily distributed Grid
grid2D=ESMF_GridCreateNoPeriDim(maxIndex=(/5,5/), &
      arbIndexList=localArbIndex, arbIndexCount=5, rc=rc)
```

To create a 3D Grid of dimensions 5x6x5 with the first and the third dimensions distributed arbitrarily, `distDim` is used.

```
! Create a 3D Grid with the 1st and 3rd dimension arbitrarily distributed
grid3D=ESMF_GridCreateNoPeriDim(maxIndex=(/5,6,5/), &
      arbIndexList=localArbIndex, arbIndexCount=5, &
      distDim=(/1,3/), rc=rc)
```

31.3.2 Create a 2D regularly distributed rectilinear Grid with uniformly spaced coordinates

The following is an example of creating a simple rectilinear grid and loading in a set of coordinates. It illustrates a straightforward use of the `ESMF_GridCreateNoPeriDim()` call described in the previous section. This code creates a 10x20 2D grid with uniformly spaced coordinates varying from (10,10) to (100,200). The grid is partitioned using a regular distribution. The first dimension is divided into two pieces, and the second dimension is divided into 3. This example assumes that the code is being run with a 1-1 mapping between PETs and DEs because we are only accessing the first DE on each PET (localDE=0). Because we have 6 DEs (2x3), this example would only work when run on 6 PETs. The Grid is created with global indices. After Grid creation the local bounds and native Fortran arrays are retrieved and the coordinates are set by the user.

```
!------------------------------------------------------------------
! Create the Grid:  Allocate space for the Grid object, define the
! topology and distribution of the Grid, and specify that it
! will have global indices.  Note that here aperiodic bounds are
! specified by the argument name. In this call the minIndex hasn't
! been set, so it defaults to (1,1,...). The default is to
! divide the index range as equally as possible among the DEs
! specified in regDecomp. This behavior can be changed by
! specifying decompFlag.
!------------------------------------------------------------------
grid2D=ESMF_GridCreateNoPeriDim(          &
      ! Define a regular distribution
      maxIndex=(/10,20/), & ! define index space
      regDecomp=(/2,3/),  & ! define how to divide among DEs
      coordSys=ESMF_COORDSYS_CART, &
      ! Specify mapping of coords dim to Grid dim
      coordDep1=(/1/), & ! 1st coord is 1D and depends on 1st Grid dim
      coordDep2=(/2/), & ! 2nd coord is 1D and depends on 2nd Grid dim
      indexflag=ESMF_INDEX_GLOBAL, &
      rc=rc)



!------------------------------------------------------------------
! Allocate coordinate storage and associate it with the center
! stagger location.  Since no coordinate values are specified in
! this call no coordinate values are set yet.
!------------------------------------------------------------------
call ESMF_GridAddCoord(grid2D,  &
       staggerloc=ESMF_STAGGERLOC_CENTER, rc=rc)
```

```
!------------------------------------------------------------------------
! Get the pointer to the first coordinate array and the bounds
! of its global indices on the local DE.
!------------------------------------------------------------------------
call ESMF_GridGetCoord(grid2D, coordDim=1, localDE=0, &
       staggerloc=ESMF_STAGGERLOC_CENTER, &
       computationalLBound=lbnd, computationalUBound=ubnd, &
       farrayPtr=coordX, rc=rc)



!------------------------------------------------------------------------
! Calculate and set coordinates in the first dimension [10-100].
!------------------------------------------------------------------------
do i=lbnd(1),ubnd(1)
     coordX(i) = i*10.0
enddo

!------------------------------------------------------------------------
! Get the pointer to the second coordinate array and the bounds of
! its global indices on the local DE.
!------------------------------------------------------------------------
call ESMF_GridGetCoord(grid2D, coordDim=2, localDE=0, &
       staggerloc=ESMF_STAGGERLOC_CENTER, &
       computationalLBound=lbnd, computationalUBound=ubnd, &
       farrayPtr=coordY, rc=rc)



!------------------------------------------------------------------------
! Calculate and set coordinates in the second dimension [10-200]
!------------------------------------------------------------------------
do j=lbnd(1),ubnd(1)
     coordY(j) = j*10.0
enddo
```

31.3.3 Create a periodic 2D regularly distributed rectilinear Grid

The following is an example of creating a simple rectilinear grid with a periodic dimension and loading in a set of coordinates. It illustrates a straightforward use of the `ESMF_GridCreate1PeriDim()` call described in the previous section. This code creates a 360x180 2D grid with uniformly spaced coordinates varying from (1,1) to (360,180). The grid is partitioned using a regular distribution. The first dimension is divided into two pieces, and the second dimension is divided into 3. This example assumes that the code is being run with a 1-1 mapping between PETs and DEs because we are only accessing the first DE on each PET (localDE=0). Because we have 6 DEs (2x3), this example would only work when run on 6 PETs. The Grid is created with global indices. After Grid creation the local bounds and native Fortran arrays are retrieved and the coordinates are set by the user.

```
!------------------------------------------------------------------------
! Create the Grid:  Allocate space for the Grid object, define the
! topology and distribution of the Grid, and specify that it
```

```
! will have global indices.  Note that here a single periodic connection
! is specified by the argument name. In this call the minIndex hasn't
! been set, so it defaults to (1,1,...). The default is to
! divide the index range as equally as possible among the DEs
! specified in regDecomp. This behavior can be changed by
! specifying decompFlag. Since the coordinate system is
! not specified, it defaults to ESMF_COORDSYS_SPH_DEG.
!------------------------------------------------------------------
grid2D=ESMF_GridCreate1PeriDim(          &
      ! Define a regular distribution
      maxIndex=(/360,180/), & ! define index space
      regDecomp=(/2,3/),  & ! define how to divide among DEs
      ! Specify mapping of coords dim to Grid dim
      coordDep1=(/1/), & ! 1st coord is 1D and depends on 1st Grid dim
      coordDep2=(/2/), & ! 2nd coord is 1D and depends on 2nd Grid dim
      indexflag=ESMF_INDEX_GLOBAL, &
      rc=rc)




!------------------------------------------------------------------
! Allocate coordinate storage and associate it with the center
! stagger location.  Since no coordinate values are specified in
! this call no coordinate values are set yet.
!------------------------------------------------------------------
call ESMF_GridAddCoord(grid2D,  &
       staggerloc=ESMF_STAGGERLOC_CENTER, rc=rc)




!------------------------------------------------------------------
! Get the pointer to the first coordinate array and the bounds
! of its global indices on the local DE.
!------------------------------------------------------------------
call ESMF_GridGetCoord(grid2D, coordDim=1, localDE=0, &
       staggerloc=ESMF_STAGGERLOC_CENTER, &
       computationalLBound=lbnd, computationalUBound=ubnd, &
       farrayPtr=coordX, rc=rc)




!------------------------------------------------------------------
! Calculate and set coordinates in the first dimension [10-100].
!------------------------------------------------------------------
do i=lbnd(1),ubnd(1)
     coordX(i) = i*1.0
enddo

!------------------------------------------------------------------
! Get the pointer to the second coordinate array and the bounds of
! its global indices on the local DE.
```

```
!------------------------------------------------------------------
call ESMF_GridGetCoord(grid2D, coordDim=2, localDE=0, &
       staggerloc=ESMF_STAGGERLOC_CENTER, &
       computationalLBound=lbnd, computationalUBound=ubnd, &
       farrayPtr=coordY, rc=rc)




!------------------------------------------------------------------
! Calculate and set coordinates in the second dimension [10-200]
!------------------------------------------------------------------
do j=lbnd(1),ubnd(1)
     coordY(j) = j*1.0
enddo
```

The remaining examples in this section will use the irregular distribution because of its greater generality. To create code similar to these, but using a regular distribution, replace the `countsPerDEDim` arguments in the Grid create with the appropriate `maxIndex` and `regDecomp` arguments.

31.3.4 Create a 2D irregularly distributed rectilinear Grid with uniformly spaced coordinates

This example serves as an illustration of the difference between using a regular and irregular distribution. It repeats the previous example except using an irregular distribution to give the user more control over how the cells are divided between the DEs. As before, this code creates a 10x20 2D Grid with uniformly spaced coordinates varying from (10,10) to (100,200). In this example, the Grid is partitioned using an irregular distribution. The first dimension is divided into two pieces, the first with 3 Grid cells per DE and the second with 7 Grid cells per DE. In the second dimension, the Grid is divided into 3 pieces, with 11, 2, and 7 cells per DE respectively. This example assumes that the code is being run with a 1-1 mapping between PETs and DEs because we are only accessing the first DE on each PET (localDE=0). Because we have 6 DEs (2x3), this example would only work when run on 6 PETs. The Grid is created with global indices. After Grid creation the local bounds and native Fortran arrays are retrieved and the coordinates are set by the user.

```
!------------------------------------------------------------------
! Create the Grid:  Allocate space for the Grid object, define the
! topology and distribution of the Grid, and specify that it
! will have global coordinates.  Note that aperiodic bounds are
! indicated by the method name. In this call the minIndex hasn't
! been set, so it defaults to (1,1,...).
!------------------------------------------------------------------
grid2D=ESMF_GridCreateNoPeriDim(          &
         ! Define an irregular distribution
         countsPerDEDim1=(/3,7/),    &
         countsPerDEDim2=(/11,2,7/), &
         ! Specify mapping of coords dim to Grid dim
         coordDep1=(/1/), & ! 1st coord is 1D and depends on 1st Grid dim
         coordDep2=(/2/), & ! 2nd coord is 1D and depends on 2nd Grid dim
         indexflag=ESMF_INDEX_GLOBAL, &
         rc=rc)
```

```
!------------------------------------------------------------------
! Allocate coordinate storage and associate it with the center
! stagger location.  Since no coordinate values are specified in
! this call no coordinate values are set yet.
!------------------------------------------------------------------
call ESMF_GridAddCoord(grid2D,  &
       staggerloc=ESMF_STAGGERLOC_CENTER, rc=rc)



!------------------------------------------------------------------
! Get the pointer to the first coordinate array and the bounds
! of its global indices on the local DE.
!------------------------------------------------------------------
call ESMF_GridGetCoord(grid2D, coordDim=1, localDE=0, &
       staggerloc=ESMF_STAGGERLOC_CENTER, &
       computationalLBound=lbnd, computationalUBound=ubnd, &
       farrayPtr=coordX, rc=rc)



!------------------------------------------------------------------
! Calculate and set coordinates in the first dimension [10-100].
!------------------------------------------------------------------
do i=lbnd(1),ubnd(1)
     coordX(i) = i*10.0
enddo

!------------------------------------------------------------------
! Get the pointer to the second coordinate array and the bounds of
! its global indices on the local DE.
!------------------------------------------------------------------
call ESMF_GridGetCoord(grid2D, coordDim=2, localDE=0, &
       staggerloc=ESMF_STAGGERLOC_CENTER, &
       computationalLBound=lbnd, computationalUBound=ubnd, &
       farrayPtr=coordY, rc=rc)



!------------------------------------------------------------------
! Calculate and set coordinates in the second dimension [10-200]
!------------------------------------------------------------------
do j=lbnd(1),ubnd(1)
     coordY(j) = j*10.0
enddo
```

31.3.5 Create a 2D irregularly distributed Grid with curvilinear coordinates

The following is an example of creating a simple curvilinear Grid and loading in a set of coordinates. It creates a 10x20 2D Grid where the coordinates vary along every dimension. The Grid is partitioned using an irregular distribution. The

first dimension is divided into two pieces, the first with 3 Grid cells per DE and the second with 7 Grid cells per DE. In the second dimension, the Grid is divided into 3 pieces, with 11, 2, and 7 cells per DE respectively. This example assumes that the code is being run with a 1-1 mapping between PETs and DEs because we are only accessing the first DE on each PET (localDE=0). Because we have 6 DEs (2x3), this example would only work when run on 6 PETs. The Grid is created with global indices. After Grid creation the local bounds and native Fortran arrays are retrieved and the coordinates are set by the user.

```
!------------------------------------------------------------------------
! Create the Grid:  Allocate space for the Grid object, define the
! distribution of the Grid, and specify that it
! will have global indices.  Note that aperiodic bounds are
! indicated by the method name. If periodic bounds were desired they
! could be specified by using the ESMF_GridCreate1PeriDim() call.
! In this call the minIndex hasn't been set, so it defaults to (1,1,...).
!------------------------------------------------------------------------
grid2D=ESMF_GridCreateNoPeriDim(      &
     ! Define an irregular distribution
     countsPerDEDim1=(/3,7/),     &
     countsPerDEDim2=(/11,2,7/),   &
     ! Specify mapping of coords dim to Grid dim
     coordDep1=(/1,2/), & ! 1st coord is 2D and depends on both Grid dim
     coordDep2=(/1,2/), & ! 2nd coord is 2D and depends on both Grid dim
     indexflag=ESMF_INDEX_GLOBAL, &
     rc=rc)




!------------------------------------------------------------------------
! Allocate coordinate storage and associate it with the center
! stagger location.  Since no coordinate values are specified in
! this call no coordinate values are set yet.
!------------------------------------------------------------------------
call ESMF_GridAddCoord(grid2D,  &
       staggerloc=ESMF_STAGGERLOC_CENTER, rc=rc)




!------------------------------------------------------------------------
! Get the pointer to the first coordinate array and the bounds
! of its global indices on the local DE.
!------------------------------------------------------------------------
call ESMF_GridGetCoord(grid2D, coordDim=1, localDE=0, &
       staggerloc=ESMF_STAGGERLOC_CENTER, &
       computationalLBound=lbnd, computationalUBound=ubnd, &
       farrayPtr=coordX2D, rc=rc)




!------------------------------------------------------------------------
! Calculate and set coordinates in the first dimension [10-100].
!------------------------------------------------------------------------
```

```
do j=lbnd(2),ubnd(2)
do i=lbnd(1),ubnd(1)
     coordX2D(i,j) = i+j
enddo
enddo

!------------------------------------------------------------------------
! Get the pointer to the second coordinate array and the bounds of
! its global indices on the local DE.
!------------------------------------------------------------------------
call ESMF_GridGetCoord(grid2D, coordDim=2, localDE=0, &
       staggerloc=ESMF_STAGGERLOC_CENTER, &
       computationalLBound=lbnd, computationalUBound=ubnd, &
       farrayPtr=coordY2D, rc=rc)




!------------------------------------------------------------------------
! Calculate and set coordinates in the second dimension [10-200]
!------------------------------------------------------------------------
do j=lbnd(2),ubnd(2)
do i=lbnd(1),ubnd(1)
     coordY2D(i,j) = j-i/100.0
enddo
enddo
```

31.3.6 Create an irregularly distributed rectilinear Grid with a non-distributed vertical dimension

This example demonstrates how a user can build a rectilinear horizontal Grid with a non-distributed vertical dimension. The Grid contains both the center and corner stagger locations (i.e. Arakawa B-Grid). In contrast to the previous examples, this example doesn't assume that the code is being run with a 1-1 mapping between PETs and DEs. It should work when run on any number of PETs.

```
!------------------------------------------------------------------------
! Create the Grid:  Allocate space for the Grid object.  The
! Grid is defined to be 180 Grid cells in the first dimension
! (e.g. longitude), 90 Grid cells in the second dimension
! (e.g. latitude), and 40 Grid cells in the third dimension
! (e.g. height).  The first dimension is decomposed over 4 DEs,
! the second over 3 DEs, and the third is not distributed.
! The connectivities in each dimension are set to aperiodic
! by this method. In this call the minIndex hasn't been set,
! so it defaults to (1,1,...).
!------------------------------------------------------------------------
grid3D=ESMF_GridCreateNoPeriDim( &
         ! Define an irregular distribution
         countsPerDEDim1=(/45,75,40,20/), &
         countsPerDEDim2=(/30,40,20/),    &
         countsPerDEDim3=(/40/),          &
         ! Specify mapping of coords dim to Grid dim
         coordDep1=(/1/), & ! 1st coord is 1D and depends on 1st Grid dim
         coordDep2=(/2/), & ! 2nd coord is 1D and depends on 2nd Grid dim
```

```
          coordDep3=(/3/), & ! 3rd coord is 1D and depends on 3rd Grid dim
          indexflag=ESMF_INDEX_GLOBAL,     & ! Use global indices
          rc=rc)



!------------------------------------------------------------------------
! Allocate coordinate storage for both center and corner stagger
! locations.  Since no coordinate values are specified in this
! call no coordinate values are set yet.
!------------------------------------------------------------------------
call ESMF_GridAddCoord(grid3D, &
       staggerloc=ESMF_STAGGERLOC_CENTER_VCENTER, rc=rc)



call ESMF_GridAddCoord(grid3D, &
       staggerloc=ESMF_STAGGERLOC_CORNER_VCENTER, rc=rc)




!------------------------------------------------------------------------
! Get the number of DEs on this PET, so that the program
! can loop over them when accessing data.
!------------------------------------------------------------------------
call ESMF_GridGet(grid3D, localDECount=localDECount, rc=rc)




!------------------------------------------------------------------------
! Loop over each localDE when accessing data
!------------------------------------------------------------------------
do lDE=0,localDECount-1

 !-----------------------------------------------------------------------
 ! Fill in the coordinates for the corner stagger location first.
 !-----------------------------------------------------------------------
   !----------------------------------------------------------------------
   ! Get the local bounds of the global indexing for the first
   ! coordinate array on the local DE. If the number of PETs
   ! is less than the total number of DEs then the rest of this
   ! example would be in a loop over the local DEs.  Also get the
   ! pointer to the first coordinate array.
   !----------------------------------------------------------------------
   call ESMF_GridGetCoord(grid3D, coordDim=1, localDE=lDE, &
          staggerLoc=ESMF_STAGGERLOC_CORNER_VCENTER,       &
          computationalLBound=lbnd_corner,                 &
          computationalUBound=ubnd_corner,                 &
          farrayPtr=cornerX, rc=rc)
```

```
  !-------------------------------------------------------------------
  ! Calculate and set coordinates in the first dimension.
  !-------------------------------------------------------------------
  do i=lbnd_corner(1),ubnd_corner(1)
     cornerX(i) = (i-1)*(360.0/180.0)
  enddo

  !-------------------------------------------------------------------
  ! Get the local bounds of the global indexing for the second
  ! coordinate array on the local DE.  Also get the pointer to the
  ! second coordinate array.
  !-------------------------------------------------------------------
  call ESMF_GridGetCoord(grid3D, coordDim=2, localDE=lDE,   &
         staggerLoc=ESMF_STAGGERLOC_CORNER_VCENTER,         &
         computationalLBound=lbnd_corner,                   &
         computationalUBound=ubnd_corner,                   &
         farrayPtr=cornerY, rc=rc)




  !-------------------------------------------------------------------
  ! Calculate and set coordinates in the second dimension.
  !-------------------------------------------------------------------
  do j=lbnd_corner(1),ubnd_corner(1)
     cornerY(j) = (j-1)*(180.0/90.0)
  enddo

  !-------------------------------------------------------------------
  ! Get the local bounds of the global indexing for the third
  ! coordinate array on the local DE, and the pointer to the array.
  !-------------------------------------------------------------------
  call ESMF_GridGetCoord(grid3D, coordDim=3, localDE=lDE,   &
         staggerloc=ESMF_STAGGERLOC_CENTER_VCENTER,         &
         computationalLBound=lbnd, computationalUBound=ubnd,&
         farrayPtr=cornerZ, rc=rc)




  !-------------------------------------------------------------------
  ! Calculate and set the vertical coordinates
  !-------------------------------------------------------------------
  do k=lbnd(1),ubnd(1)
     cornerZ(k) = 4000.0*( (1./39.)*(k-1)  )**2
  enddo

!-------------------------------------------------------------------
! Now fill the coordinates for the center stagger location with
! the average of the corner coordinate location values.
!-------------------------------------------------------------------
  !-------------------------------------------------------------------
  ! Get the local bounds of the global indexing for the first
```

```
! coordinate array on the local DE, and the pointer to the array.
!------------------------------------------------------------------
call ESMF_GridGetCoord(grid3D, coordDim=1, localDE=lDE,    &
       staggerloc=ESMF_STAGGERLOC_CENTER_VCENTER,          &
       computationalLBound=lbnd, computationalUBound=ubnd, &
       farrayPtr=centerX, rc=rc)




!------------------------------------------------------------------
! Calculate and set coordinates in the first dimension.
!------------------------------------------------------------------
do i=lbnd(1),ubnd(1)
   centerX(i) = 0.5*(i-1 + i)*(360.0/180.0)
enddo

!------------------------------------------------------------------
! Get the local bounds of the global indexing for the second
! coordinate array on the local DE, and the pointer to the array.
!------------------------------------------------------------------
 call ESMF_GridGetCoord(grid3D, coordDim=2, localDE=lDE,    &
        staggerloc=ESMF_STAGGERLOC_CENTER_VCENTER,          &
        computationalLBound=lbnd, computationalUBound=ubnd, &
        farrayPtr=centerY, rc=rc)




!------------------------------------------------------------------
! Calculate and set coordinates in the second dimension.
!------------------------------------------------------------------
do j=lbnd(1),ubnd(1)
   centerY(j) = 0.5*(j-1 + j)*(180.0/90.0)
enddo

!------------------------------------------------------------------
! Get the local bounds of the global indexing for the third
! coordinate array on the local DE, and the pointer to the array.
!------------------------------------------------------------------
call ESMF_GridGetCoord(grid3D, coordDim=3, localDE=lDE,   &
       staggerloc=ESMF_STAGGERLOC_CENTER_VCENTER,         &
       computationalLBound=lbnd, computationalUBound=ubnd,&
       farrayPtr=centerZ, rc=rc)




!------------------------------------------------------------------
! Calculate and set the vertical coordinates
!------------------------------------------------------------------
do k=lbnd(1),ubnd(1)
   centerZ(k) = 4000.0*( (1./39.)*(k-1)  )**2
enddo
```

```
!------------------------------------------------------------------
! End of loop over DEs
!------------------------------------------------------------------
enddo
```

31.3.7 Create an arbitrarily distributed rectilinear Grid with a non-distributed vertical dimension

There are more restrictions in defining an arbitrarily distributed grid. First, there is always one DE per PET. Secondly, only local index (ESMF_INDEX_LOCAL) is supported. Third, only one stagger location, i.e. ESMF_STAGGERLOC_CENTER is allowed and last there is no extra paddings on the edge of the grid.

This example demonstrates how a user can build a 3D grid with its rectilinear horizontal Grid distributed arbitrarily and a non-distributed vertical dimension.

```
!------------------------------------------------------------------
! Set up the local index array:  Assuming the grid is 360x180x10.  First
! calculate the localArbIndexCount and localArbIndex array for each PET
! based on the total number of PETs. The cells are evenly distributed in
! all the PETs. If the total number of cells are not divisible by the
! total PETs, the remaining cells are assigned to the last PET.  The
! cells are card dealt to each PET in y dimension first,
! i.e. (1,1) -> PET 0, (1,2)->  PET 1, (1,3)-> PET 2, and so forth.
!------------------------------------------------------------------
xdim = 360
ydim = 180
zdim = 10
localArbIndexCount = (xdim*ydim)/petCount
remain = (xdim*ydim)-localArbIndexCount*petCount
if (localPet == petCount-1) localArbIndexCount = localArbIndexCount+remain

allocate(localArbIndex(localArbIndexCount,2))
ind = localPet
do i=1, localArbIndexCount
   localArbIndex(i,1)=mod(ind,ydim)+1
   localArbIndex(i,2)=ind/ydim + 1
   ind = ind + petCount
enddo
if (localPet == petCount-1) then
   ind = xdim*ydim-remain+1
   do i=localArbIndexCount-remain+1,localArbIndexCount
      localArbIndex(i,1)=mod(ind,ydim)+1
      localArbIndex(i,2)=ind/ydim+1
      ind = ind + 1
   enddo
endif

!------------------------------------------------------------------
! Create the Grid:  Allocate space for the Grid object.
! the minIndex hasn't been set, so it defaults to (1,1,...). The
! default coordDep1 and coordDep2 are (/ESMF_DIM_ARB/) where
```

```
! ESMF_DIM_ARB represents the collapsed dimension for the
! arbitrarily distributed grid dimensions.  For the undistributed
! grid dimension, the default value for coordDep3 is (/3/).  The
! default values for coordDepX in the arbitrary distribution are
! different from the non-arbitrary distributions.
!------------------------------------------------------------------
grid3D=ESMF_GridCreateNoPeriDim( &
         maxIndex = (/xdim, ydim, zdim/), &
         arbIndexList = localArbIndex, &
         arbIndexCount = localArbIndexCount, &
         rc=rc)




!------------------------------------------------------------------
! Allocate coordinate storage for the center stagger location, the
! only stagger location supported for the arbitrary distribution.
!------------------------------------------------------------------
call ESMF_GridAddCoord(grid3D, &
       staggerloc=ESMF_STAGGERLOC_CENTER_VCENTER, rc=rc)




!------------------------------------------------------------------
! Fill in the coordinates for the center stagger location. There is
! always one DE per PET, so localDE is always 0
!------------------------------------------------------------------
call ESMF_GridGetCoord(grid3D, coordDim=1, localDE=0, &
       staggerLoc=ESMF_STAGGERLOC_CENTER,       &
       computationalLBound=lbnd,                 &
       computationalUBound=ubnd,                 &
       farrayPtr=centerX, rc=rc)




!------------------------------------------------------------------
! Calculate and set coordinates in the first dimension.
!------------------------------------------------------------------
do i=lbnd(1),ubnd(1)
   centerX(i) = (localArbIndex(i,1)-0.5)*(360.0/xdim)
enddo


!------------------------------------------------------------------
! Get the local bounds of the global indexing for the second
! coordinate array on the local DE, and the pointer to the array.
!------------------------------------------------------------------
call ESMF_GridGetCoord(grid3D, coordDim=2, localDE=0,    &
       staggerloc=ESMF_STAGGERLOC_CENTER,                  &
       computationalLBound=lbnd, computationalUBound=ubnd, &
```

```
       farrayPtr=centerY, rc=rc)



!------------------------------------------------------------------
! Calculate and set coordinates in the second dimension.
!------------------------------------------------------------------
do j=lbnd(1),ubnd(1)
   centerY(j) = (localArbIndex(j,2)-0.5)*(180.0/ydim)-90.0
enddo

!------------------------------------------------------------------
! Get the local bounds of the global indexing for the third
! coordinate array on the local DE, and the pointer to the array.
!------------------------------------------------------------------
call ESMF_GridGetCoord(grid3D, coordDim=3, localDE=0,   &
       staggerloc=ESMF_STAGGERLOC_CENTER,               &
       computationalLBound=lbnd, computationalUBound=ubnd,&
       farrayPtr=centerZ, rc=rc)




!------------------------------------------------------------------
! Calculate and set the vertical coordinates
!------------------------------------------------------------------
do k=lbnd(1),ubnd(1)
   centerZ(k) = 4000.0*( (1./zdim)*(k-1))**2
enddo
```

31.3.8 Create a curvilinear Grid using the coordinates defined in a SCRIP file

ESMF supports the creation of a 2D curvilinear Grid using the coordinates defined in a SCRIP format Grid file [**?**]. The grid contained in the file must be a 2D logically rectangular grid with `grid_rank` in the file set to 2. The center coordinates variables `grid_center_lat` and `grid_center_lon` in the file are placed in the ESMF_STAGGERLOC_CENTER location. If the parameter `addCornerStagger` in the `ESMF_GridCreate` call is set to .true., then the variables `grid_corner_lat` and `grid_corner_lon` in the file are used to set the ESMF_STAGGERLOC_CORNER coordinates, otherwise they are ignored. The values in the `grid_imask` variable in the file are used to set the ESMF_GRIDITEM_MASK in the Grid.

The following example code shows you how to create a 2D Grid with both center and corner coordinates using a SCRIP file and a row only regular distribution:

```
grid2D = ESMF_GridCreate(filename="data/T42_grid.nc",  &
           fileFormat=ESMF_FILEFORMAT_SCRIP,  &
           regDecomp=(/PetCount,1/), addCornerStagger=.true., rc=rc)
```

where T42_grid.nc is a 2D global grid of size (128x64) and the resulting Grid is distributed by partitioning the rows evenly over all the PETs.

ESMF also support the creation of a 2D Grid from the SCRIP format Grid file using a user specified ESMF_DistGrid. The following example code demonstrates the creation of an Grid object using a pre-defined DistGrid. The resulting Grid is the same as the one created above:

```
   distgrid = ESMF_DistGridCreate((/1,1/), (/128,64/), &
              regDecomp=(/PetCount,1/), rc=rc)
   grid2D = ESMF_GridCreate(filename="data/T42_grid.nc",  &
              fileFormat=ESMF_FILEFORMAT_SCRIP,  &
              distGrid=distgrid, addCornerStagger=.true., rc=rc)
```

31.3.9 Create an empty Grid in a parent Component for completion in a child Component

ESMF Grids can be created incrementally. To do this, the user first calls `ESMF_GridEmptyCreate()` to allocate the shell of a Grid. Next, we use the `ESMF_GridEmptyComplete()` call that fills in the Grid and does an internal commit to make it usable. For consistency's sake the `ESMF_GridSetCommitShapeTile()` call must occur on the same or a subset of the PETs as the `ESMF_GridEmptyCreate()` call. The `ESMF_GridEmptyComplete()` call uses the VM for the context in which it's executed and the "empty" Grid contains no information about the VM in which its create was run. This means that if the `ESMF_GridEmptyComplete()` call occurs in a subset of the PETs in which the `ESMF_GridEmptyCreate()` was executed that the Grid is created only in that subset. Inside the subset the Grid will be fine, but outside the subset the Grid objects will still be "empty" and not usable. The following example uses the incremental technique to create a rectangular 10x20 Grid with coordinates at the center and corner stagger locations.

```
!------------------------------------------------------------------------
! IN THE PARENT COMPONENT:
! Create an empty Grid in the parent component for use in a child component.
! The parent may be defined on more PETs than the child component.
! The child's [vm or pet list] is passed into the create call so that
! the Grid is defined on the appropriate subset of the parent's PETs.
!------------------------------------------------------------------------
   grid2D=ESMF_GridEmptyCreate(rc=rc)




!------------------------------------------------------------------------
! IN THE CHILD COMPONENT:
! Set the Grid topology.  Here we define an irregularly distributed
! rectangular Grid.
!------------------------------------------------------------------------
   call ESMF_GridEmptyComplete(grid2D,             &
                         countsPerDEDim1=(/6,4/),      &
                         countsPerDEDim2=(/10,3,7/), rc=rc)




!------------------------------------------------------------------------
! Add Grid coordinates at the cell center location.
!------------------------------------------------------------------------
   call ESMF_GridAddCoord(grid2D, staggerLoc=ESMF_STAGGERLOC_CENTER, rc=rc)
```

```
!------------------------------------------------------------------------
! Add Grid coordinates at the corner stagger location.
!------------------------------------------------------------------------
   call ESMF_GridAddCoord(grid2D, staggerLoc=ESMF_STAGGERLOC_CORNER, rc=rc)
```

31.3.10 Create a six-tile cubed sphere Grid

This example creates a multi-tile Grid to represent a cubed sphere grid. Each of the six tiles making up the cubed sphere has 45 elements on each side, so the total number of elements is 45x45x6=12150. Each tile is decomposed using a regular decomposition. The first two tiles are decomposed into 2x2 blocks each and the remaining 4 tiles are decomposed into 1x2 block. A total of 16 DEs are used.

In this example, both the center and corner coordinates will be added to the grid.

```
! Set up decomposition for each tile
allocate(decomptile(2,6))
decomptile(:,1)=(/2,2/) ! Tile 1
decomptile(:,2)=(/2,2/) ! Tile 2
decomptile(:,3)=(/1,2/) ! Tile 3
decomptile(:,4)=(/1,2/) ! Tile 4
decomptile(:,5)=(/1,2/) ! Tile 5
decomptile(:,6)=(/1,2/) ! Tile 6

! Create cubed sphere grid
grid2D = ESMF_GridCreateCubedSphere(tileSize=45, regDecompPTile=decomptile, &
            staggerLocList=(/ESMF_STAGGERLOC_CENTER, ESMF_STAGGERLOC_CORNER/), rc=rc)
```

31.3.11 Create a six-tile cubed sphere Grid and apply Schmidt transform

This example creates the same cubed sphere grid with the same regular decomposition as in 31.3.10 with a few differences. First, the coordinates of the grid are of type `ESMF_TYPEKIND_R4` instead of the default `ESMF_TYPEKIND_R8`. Secondly, the coordinate system is `ESMF_COORDSYS_SPH_RAD` instead of the default `ESMF_COORDSYS_SPH_DEG`. Lastly, the grid was then transformed using Schmidt Transformation algorithm on an arbitrary target point and a streatching factor. An optional argument `TransformArgs` of type `ESMF_CubedSphereTransform_Args` is used to pass the Schmidt Transform arguments. `ESMF_CubedSphereTransform_Args` is defined as follows:

```
type ESMF_CubedSphereTransform_Args
   real(ESMF_KIND_R4) :: stretch_factor, target_lat, target_lon
end type
```

Note `target_lat` and `target_lon` are in radians.

```
transformArgs%stretch_factor = 0.5;
transformArgs%target_lat = 0.0; ! in radians
transformArgs%target_lat = 1.3; ! in radians
grid2D = ESMF_GridCreateCubedSphere(tileSize=45, regDecompPTile=decomptile, &
```

```
                staggerLocList = (/ESMF_STAGGERLOC_CENTER, ESMF_STAGGERLOC_CORNER/), &
                coordTypeKind = ESMF_TYPEKIND_R4, &
                coordSys = ESMF_COORDSYS_SPH_RAD, &
                transformArgs=transformArgs, &
                rc=rc)
```

31.3.12 Create a six-tile cubed sphere Grid from a GRIDSPEC Mosaic file

This example creates a six-tile Grid to represent a cubed sphere grid defined in a GRIDSPEC Mosaic file `C48_mosaic.nc`. The GRIDSPEC mosaic file format is defined in the document GRIDSPEC: A standard for the description of grids used in Earth System models by V. Balaji, Alistair Adcroft and Zhi Liang.

The mosaic file contains the following information:

```
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) ;
              gridlocation:standard_name = "grid_file_location" ;
       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" ;

// global attributes:
              :grid_version = "0.2" ;
              :code_version = "$Name: testing $" ;
data:

mosaic = "C48_mosaic" ;

gridlocation = "/archive/z1l/tools/test_20091028/output_all/" ;

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 dummy variable with its `standard_name` attribute set to `grid_mosaic_spec` is required. 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 has three required attributes: `standard_name`, `contact_type`, and `contact_index`. `startand_name` has to be set to `grid_contact_spec`. `contact_type` has to be `boundary`. ESMF does not support overlapping contact regions. `contact_index` defines the variable name that contains the information how the two adjacent tiles are connected to each other. 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 tile2. 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
```

The associated tile file names are defined in variable `gridfiles` and the directory path is defined in variable `gridlocation`. The `gridlocation` can be overwritten with an optional arguemnt `TileFilePath`. Each tile is decomposed using a regular decomposition. The first two tiles are decomposed into 2x2 blocks each and the remaining 4 tiles are decomposed into 1x2 block. A total of 16 DEs are used.

`ESMF_GridCreateMosaic()` first reads in the mosaic file and defines the tile connections in the `ESMF_DistGrid` using the information defined in variables `contacts` and `contact_index`. Then it reads in the coordinates defined in the tile files if the optional argument `staggerLocList` is provided. The coordinates defined in the tile file are a `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" ;
               :code_version = "$Name: testing $" ;
               :history = "/home/z1l/bin/tools_20091028/make_hgrid --grid_type gnomonic_e
}
```

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 currently only use the `geographic_longitude` and `geographic_latitude` variables.

```
    ! Set up decomposition for each tile
    allocate(decomptile(2,6))
    decomptile(:,1)=(/2,2/) ! Tile 1
    decomptile(:,2)=(/2,2/) ! Tile 2
    decomptile(:,3)=(/1,2/) ! Tile 3
    decomptile(:,4)=(/1,2/) ! Tile 4
    decomptile(:,5)=(/1,2/) ! Tile 5
    decomptile(:,6)=(/1,2/) ! Tile 6

    ! Create cubed sphere grid without reading in the coordinates
    grid2D = ESMF_GridCreateMosaic(filename='data/C48_mosaic.nc', &
               tileFilePath='./data/', regDecompPTile=decomptile, rc=rc)



    ! Create cubed sphere grid and read in the center and corner stagger coordinates
    ! from the tile files

    grid2D = ESMF_GridCreateMosaic(filename='data/C48_mosaic.nc', &
               staggerLocList=(/ESMF_STAGGERLOC_CENTER, ESMF_STAGGERLOC_CORNER/), &
               tileFilePath='./data/', regDecompPTile=decomptile, rc=rc)




    ! Create cubed sphere grid and read in the edge staggers' coordinates
    ! from the tile files, set the coordTypeKind to ESMF_TYPEKIND_R4

    grid2D = ESMF_GridCreateMosaic(filename='data/C48_mosaic.nc', &
               staggerLocList=(/ESMF_STAGGERLOC_EDGE1, ESMF_STAGGERLOC_EDGE2/), &
               coordTypeKind = ESMF_TYPEKIND_R4, &
               tileFilePath='./data/', regDecompPTile=decomptile, rc=rc)
```

31.3.13 Grid stagger locations

A useful finite difference technique is to place different physical quantities at different locations within a grid cell. This *staggering* of the physical variables on the mesh is introduced so that the difference of a field is naturally defined at the location of another variable. This method was first formalized by Mesinger and Arakawa (1976).

To support the staggering of variables, the Grid provides the idea of *stagger locations*. Stagger locations refer to the places in a Grid cell that can contain coordinates or other data and once a Grid is associated with a Field object, field

data. Typically Grid data can be located at the cell center, at the cell corners, or at the cell faces, in 2D, 3D, and higher dimensions. (Note that any Arakawa stagger can be constructed of a set of Grid stagger locations.) There are predefined stagger locations (see Section 31.2.6), or, should the user wish to specify their own, there is also a set of methods for generating custom locations (See Section 31.3.25). Users can put Grid data (e.g. coordinates) at multiple stagger locations in a Grid. In addition, the user can create a Field at any of the stagger locations in a Grid.

By default the Grid data array at the center stagger location starts at the bottom index of the Grid (default (1,1..,1)) and extends up to the maximum cell index in the Grid (e.g. given by the `maxIndex` argument). Other stagger locations also start at the bottom index of the Grid, however, they can extend to +1 element beyond the center in some dimensions to allow for the extra space to surround the center elements. See Section 31.3.25 for a description of this extra space and how to adjust if it necessary. There are `ESMF_GridGet` subroutines (e.g. `ESMF_GridGetCoord()` or `ESMF_GridGetItem()`) which can be used to retrieve the stagger bounds for the piece of Grid data on a particular DE.

31.3.14 Associate coordinates with stagger locations

The primary type of data the Grid is responsible for storing is coordinates. The coordinate values in a Grid can be employed by the user in calculations or to describe the geometry of a Field. The Grid coordinate values are also used by `ESMF_FieldRegridStore()` when calculating the interpolation matrix between two Fields. The user can allocate coordinate arrays without setting coordinate values using the `ESMF_GridAddCoord()` call. (See Section 31.3.16 for a discussion of setting/getting coordinate values.) When adding or accessing coordinate data, the stagger location is specified to tell the Grid method where in the cell to get the data. The different stagger locations may also have slightly different index ranges and sizes. Please see Section 31.3.13 for a discussion of Grid stagger locations.

The following example adds coordinate storage to the corner stagger location in a Grid using one of the predefined stagger locations.

```
call ESMF_GridAddCoord(grid2D, staggerLoc=ESMF_STAGGERLOC_CORNER, rc=rc)
```

Note only the center stagger location `ESMF_STAGGERLOC_CENTER` is supported in an arbitrarily distributed Grid.

31.3.15 Specify the relationship of coordinate Arrays to index space dimensions

To specify how the coordinate arrays are mapped to the index dimensions the arguments `coordDep1`, `coordDep2`, and `coordDep3` are used, each of which is a Fortran array. The values of the elements in a `coordDep` array specify which index dimension the corresponding coordinate dimension maps to. For example, `coordDep1=(/1,2/)` means that the first dimension of coordinate 1 maps to index dimension 1 and the second maps to index dimension 2. For a grid with non-arbitrary distribution, the default values for `coordDep1`, `coordDep2` and `coordDep3` are `/1,2..,gridDimCount/`. This default thus specifies a curvilinear grid.

The following call demonstrates the creation of a 10x20 2D rectilinear grid where the first coordinate component is mapped to the second index dimension (i.e. is of size 20) and the second coordinate component is mapped to the first index dimension (i.e. is of size 10).

```
grid2D=ESMF_GridCreateNoPeriDim(countsPerDEDim1=(/5,5/), &
       countsPerDEDim2=(/7,7,6/),                        &
       coordDep1=(/2/),                                  &
       coordDep2=(/1/), rc=rc)
```

The following call demonstrates the creation of a 10x20x30 2D plus 1 curvilinear grid where coordinate component 1 and 2 are still 10x20, but coordinate component 3 is mapped just to the third index dimension.

```
grid2D=ESMF_GridCreateNoPeriDim(countsPerDEDim1=(/6,4/), &
       countsPerDEDim2=(/10,7,3/), countsPerDEDim3=(/30/), &
       coordDep1=(/1,2/), coordDep2=(/1,2/), &
       coordDep3=(/3/), rc=rc)
```

By default the local piece of the array on each PET starts at (1,1,..), however, the indexing for each grid coordinate array on each DE may be shifted to the global indices by using the `indexflag`. For example, the following call switches the grid to use global indices.

```
grid2D=ESMF_GridCreateNoPeriDim(countsPerDEDim1=(/6,4/), &
        countsPerDEDim2=(/10,7,3/), indexflag=ESMF_INDEX_GLOBAL, rc=rc)
```

For an arbitrarily distributed grid, the default value of a coordinate array dimension is `ESMF_DIM_ARB` if the index dimension is arbitrarily distributed and is `n` where `n` is the index dimension itself when it is not distributed. The following call is equivalent to the example in Section 31.3.7

```
grid3D=ESMF_GridCreateNoPeriDim( &
         maxIndex = (/xdim, ydim, zdim/), &
         arbIndexList = localArbIndex, &
         arbIndexCount = localArbIndexCount,          &
         coordDep1 = (/ESMF_DIM_ARB/), &
         coordDep2 = (/ESMF_DIM_ARB/), &
         coordDep3 = (/3/), &
         rc=rc)
```

The following call uses non-default `coordDep1`, `coordDep2`, and `coordDep3` to create a 3D curvilinear grid with its horizontal dimensions arbitrarily distributed.

```
grid3D=ESMF_GridCreateNoPeriDim( &
         maxIndex = (/xdim, ydim, zdim/), &
         arbIndexList = localArbIndex, &
         arbIndexCount = localArbIndexCount,          &
         coordDep1 = (/ESMF_DIM_ARB, 3/), &
         coordDep2 = (/ESMF_DIM_ARB, 3/), &
         coordDep3 = (/ESMF_DIM_ARB, 3/), &
         rc=rc)
```

31.3.16 Access coordinates

Once a Grid has been created, the user has several options to access the Grid coordinate data. The first of these, `ESMF_GridSetCoord()`, enables the user to use ESMF Arrays to set data for one stagger location across the whole Grid. For example, the following sets the coordinates in the first dimension (e.g. x) for the corner stagger location to those in the ESMF Array `arrayCoordX`.

```
call ESMF_GridSetCoord(grid2D, &
       staggerLoc=ESMF_STAGGERLOC_CORNER, &
       coordDim=1, array=arrayCoordX, rc=rc)
```

The method ESMF_GridGetCoord() allows the user to obtain a reference to an ESMF Array which contains the coordinate data for a stagger location in a Grid. The user can then employ any of the standard ESMF_Array tools to operate on the data. The following copies the coordinates from the second component of the corner and puts it into the ESMF Array arrayCoordY.

```
call ESMF_GridGetCoord(grid2D,     &
       staggerLoc=ESMF_STAGGERLOC_CORNER,     &
       coordDim=2,                            &
       array=arrayCoordY, rc=rc)
```

Alternatively, the call ESMF_GridGetCoord() gets a Fortran pointer to the coordinate data. The user can then operate on this array in the usual manner. The following call gets a reference to the Fortran array which holds the data for the second coordinate (e.g. y).

```
call ESMF_GridGetCoord(grid2D, coordDim=2, localDE=0, &
       staggerloc=ESMF_STAGGERLOC_CORNER, farrayPtr=coordY2D, rc=rc)
```

31.3.17 Associate items with stagger locations

The ESMF Grids contain the ability to store other kinds of data beyond coordinates. These kinds of data are referred to as "items". Although the user is free to use this data as they see fit, the user should be aware that this data may also be used by other parts of ESMF (e.g. the ESMF_GRIDITEM_MASK item is used in regridding). Please see Section 31.2.2 for a list of valid items.

Like coordinates items are also created on stagger locations. When adding or accessing item data, the stagger location is specified to tell the Grid method where in the cell to get the data. The different stagger locations may also have slightly different index ranges and sizes. Please see Section 31.3.13 for a discussion of Grid stagger locations. The user can allocate item arrays without setting item values using the ESMF_GridAddItem() call. (See Section 31.3.18 for a discussion of setting/getting item values.)

The following example adds mask item storage to the corner stagger location in a grid.

```
call ESMF_GridAddItem(grid2D, staggerLoc=ESMF_STAGGERLOC_CORNER, &
       itemflag=ESMF_GRIDITEM_MASK, rc=rc)
```

31.3.18 Access items

Once an item has been added to a Grid, the user has several options to access the data. The first of these, ESMF_GridSetItem(), enables the user to use ESMF Arrays to set data for one stagger location across the whole Grid. For example, the following sets the mask item in the corner stagger location to those in the ESMF Array arrayMask.

```
call ESMF_GridSetItem(grid2D,              &
       staggerLoc=ESMF_STAGGERLOC_CORNER, &
       itemflag=ESMF_GRIDITEM_MASK, array=arrayMask, rc=rc)
```

The method ESMF_GridGetItem() allows the user to get a reference to the Array which contains item data for a stagger location on a Grid. The user can then employ any of the standard ESMF_Array tools to operate on the data. The following gets the mask data from the corner and puts it into the ESMF Array arrayMask.

```
call ESMF_GridGetItem(grid2D,              &
       staggerLoc=ESMF_STAGGERLOC_CORNER, &
       itemflag=ESMF_GRIDITEM_MASK,           &
       array=arrayMask, rc=rc)
```

Alternatively, the call `ESMF_GridGetItem()` gets a Fortran pointer to the item data. The user can then operate on this array in the usual manner. The following call gets a reference to the Fortran array which holds the data for the mask data.

```
call ESMF_GridGetItem(grid2D, localDE=0,   &
       staggerloc=ESMF_STAGGERLOC_CORNER,  &
       itemflag=ESMF_GRIDITEM_MASK, farrayPtr=mask2D, rc=rc)
```

31.3.19 Grid regions and bounds

Like an Array or a Field, the index space of each stagger location in the Grid contains an exclusive region, a computational region and a total region. Please see Section 28.2.6 for an in depth description of these regions.

The exclusive region is the index space defined by the distgrid of each stagger location of the Grid. This region is the region which is owned by the DE and is the region operated on by communication methods such as `ESMF_FieldRegrid()`. The exclusive region for a stagger location is based on the exclusive region defined by the DistGrid used to create the Grid. The size of the stagger exclusive region is the index space for the Grid cells, plus the stagger padding.

The default stagger padding depends on the topology of the Grid. For an unconnected dimension the stagger padding is a width of 1 on the upper side (i.e. `gridEdgeUWidth=(1,1,1,1...)`). For a periodic dimension there is no stagger padding. By adjusting `gridEdgeLWidth` and `gridEdgeUWidth`, the user can set the stagger padding for the whole Grid and thus the exclusive region can be adjusted at will around the index space corresponding to the cells. The user can also use `staggerEdgeLWidth` and `staggerEdgeUWidth` to adjust individual stagger location padding within the Grid's padding (Please see Section 31.3.26 for further discussion of customizing the stagger padding).

Figure 17 shows an example of a Grid exclusive region for the `ESMF_STAGGERLOC_CORNER` stagger with default stagger padding. This exclusive region would be for a Grid generated by either of the following calls:

```
grid2D=ESMF_GridCreateNoPeriDim(regDecomp=(/2,4/), maxIndex=(/5,15/), &
         indexflag=ESMF_INDEX_GLOBAL, rc=rc)
```

```
grid2D=ESMF_GridCreateNoPeriDim(countsPerDEDim1=(/4,4,4,3/), &
         countsPerDEDim2=(/3,2/), indexflag=ESMF_INDEX_GLOBAL, rc=rc)
```

Each rectangle in this diagram represents a DE and the numbers along the sides are the index values of the locations in the DE. Note that the exclusive region has one extra index location in each dimension than the number of cells because of the padding for the larger corner stagger location.

The computational region is a user-settable region which can be used to distinguish a particular area for computation. The Grid doesn't currently contain functionality to let the user set the computational region so it defaults to the

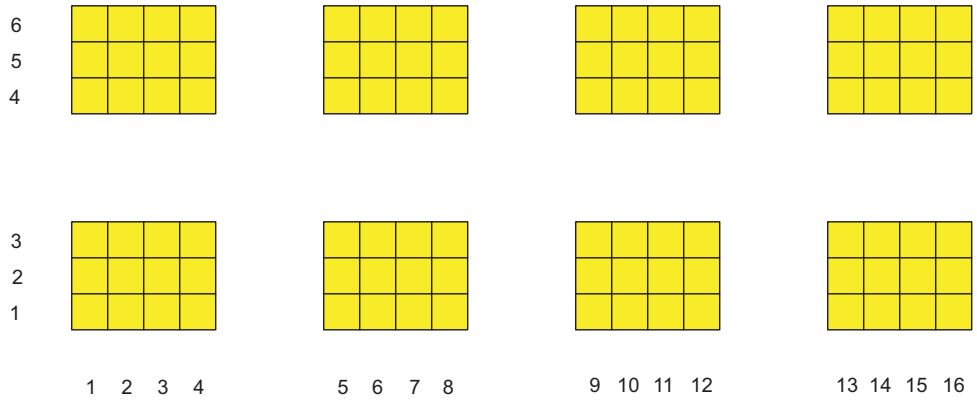


Figure 17: An example of a Grid's exclusive region for the corner stagger

exclusive region. However, if the user sets an Array holding different computational bounds into the Grid then that Array's computational bounds will be used.

The total region is the outermost boundary of the memory allocated on each DE to hold the data for the stagger location on that DE. This region can be as small as the exclusive region, but may be larger to include space for halos, memory padding, etc. The total region is what is enlarged to include space for halos, and the total region must be large enough to contain the maximum halo operation on the Grid. The Grid doesn't currently contain functionality to let the user set the total region so it defaults to the exclusive region. However, if the user sets an Array holding different total bounds into the Grid then that Array's total bounds will be used.

The user can retrieve a set of bounds for each index space region described above: exclusive bounds, computational bounds, and total bounds. Note that although some of these are similar to bounds provided by ESMF_Array subroutines (see Section 28.2.6) the format here is different. The Array bounds are only for distributed dimensions and are ordered to correspond to the dimension order in the associated DistGrid. The bounds provided by the Grid are ordered according to the order of dimensions of the data in question. This means that the bounds provided should be usable "as is" to access the data.

Each of the three types of bounds refers to the maximum and minimum per dimension of the index ranges of a particular region. The parameters referring to the maximums contain a 'U' for upper. The parameters referring to the minimums contain an 'L' for lower. The bounds and associated quantities are almost always given on a per DE basis. The three types of bounds `exclusiveBounds`, `computationalBounds`, and `totalBounds` refer to the ranges of the exclusive region, the computational region, and the total region. Each of these bounds also has a corresponding count parameter which gives the number of items across that region (on a DE) in each dimension. (e.g. `totalCount(d)=totallUBound(i)-totalLBound(i)+1`). Width parameters give the spacing between two different types of region. The `computationalWidth` argument gives the spacing between the exclusive region and the computational region. The `totalWidth` argument gives the spacing between the total region and the computational region. Like the other bound information these are typically on a per DE basis, for example specifying `totalLWidth=(1,1)` makes the bottom of the total region one lower in each dimension than the computational region on each DE. The exceptions to the per DE rule are `staggerEdgeWidth`, and `gridEdgeWidth` which give the spacing only on the DEs along the boundary of the Grid.

All the above bound discussions only apply to the grid with non-arbitrary distributions, i.e., regular or irregular distributions. For an arbitrarily distributed grid, only center stagger location is supported and there is no padding around the grid. Thus, the exclusive bounds, the total bounds and the computational bounds are identical and

staggerEdgeWidth, and gridEdgeWidth are all zeros.

31.3.20 Get Grid coordinate bounds

When operating on coordinates the user may often wish to retrieve the bounds of the piece of coordinate data on a particular local DE. This is useful for iterating through the data to set coordinates, retrieve coordinates, or do calculations. The method ESMF_GridGetCoord allows the user to retrieve bound information for a particular coordinate array.

As described in the previous section there are three types of bounds the user can get: exclusive bounds, computational bounds, and total bounds. The bounds provided by ESMF_GridGetCoordBounds are for both distributed and undistributed dimensions and are ordered according to the order of dimensions in the coordinate. This means that the bounds provided should be usable "as is" to access data in the coordinate array. In the case of factorized coordinate Arrays where a coordinate may have a smaller dimension than its associated Grid, then the dimension of the coordinate's bounds are the dimension of the coordinate, not the Grid.

The following is an example of retrieving the bounds for localDE 0 for the first coordinate array from the corner stagger location.

```
call ESMF_GridGetCoordBounds(grid2D, coordDim=1, localDE=0,  &
       staggerLoc=ESMF_STAGGERLOC_CORNER,                    &
       exclusiveLBound=elbnd, exclusiveUBound=eubnd,         &
       computationalLBound=clbnd, computationalUBound=cubnd, &
       totalLBound=tlbnd, totalUBound=tubnd, rc=rc)
```

31.3.21 Get Grid stagger location bounds

When operating on data stored at a particular stagger in a Grid the user may find it useful to be able to retrieve the bounds of the data on a particular local DE. This is useful for iterating through the data for computations or allocating arrays to hold the data. The method ESMF_GridGet allows the user to retrieve bound information for a particular stagger location.

As described in Section 31.3.19 there are three types of bounds the user can typically get, however, the Grid doesn't hold data at a stagger location (that is the job of the Field), and so no Array is contained there and so no total region exists, so the user may only retrieve exclusive and computational bounds from a stagger location. The bounds provided by ESMF_GridGet are ordered according to the order of dimensions in the Grid.

The following is an example of retrieving the bounds for localDE 0 from the corner stagger location.

```
call ESMF_GridGet(grid2D, localDE=0,                          &
       staggerLoc=ESMF_STAGGERLOC_CORNER,                     &
       exclusiveLBound=elbnd, exclusiveUBound=eubnd,          &
       computationalLBound=clbnd, computationalUBound=cubnd, rc=rc)
```

31.3.22 Get Grid stagger location information

In addition to the per DE information that can be accessed about a stagger location there is some global information that can accessed by using ESMF_GridGet without specifying a localDE. One of the uses of this information is to create an ESMF Array to hold data for a stagger location.

The information currently available from a stagger location is the distgrid. The distgrid gives the distgrid which describes the size and distribution of the elements in the stagger location.

The following is an example of retrieving information for localDE 0 from the corner stagger location.

```
! Get info about staggerloc
call ESMF_GridGet(grid2D, staggerLoc=ESMF_STAGGERLOC_CORNER,  &
       distgrid=staggerDistgrid, &
       rc=rc)
```

31.3.23 Create an Array at a stagger location

In order to create an Array to correspond to a Grid stagger location several pieces of information need to be obtained from both the Grid and the stagger location in the Grid.

The information that needs to be obtained from the Grid is the `distgridToGridMap` to ensure that the new Array has its dimensions are mapped correctly to the Grid. These are obtained using the `ESMF_GridGet` method.

The information that needs to be obtained from the stagger location is the distgrid that describes the size and distribution of the elements in the stagger location. This information can be obtained using the stagger location specific `ESMF_GridGet` method.

The following is an example of using information from a 2D Grid with non-arbitrary distribution to create an Array corresponding to a stagger location.

```
! Get info from Grid
call ESMF_GridGet(grid2D, distgridToGridMap=distgridToGridMap, rc=rc)
```

```
! Get info about staggerloc
call ESMF_GridGet(grid2D, staggerLoc=ESMF_STAGGERLOC_CORNER, &
       distgrid=staggerDistgrid, &
       rc=rc)
```

```
! construct ArraySpec
call ESMF_ArraySpecSet(arrayspec, rank=2, typekind=ESMF_TYPEKIND_R8, rc=rc)
```

```
! Create an Array based on info from grid
array=ESMF_ArrayCreate(arrayspec=arrayspec, &
        distgrid=staggerDistgrid, distgridToArrayMap=distgridToGridMap, &
        rc=rc)
```

Creating an Array for a Grid with arbitrary distribution is different. For a 2D Grid with both dimension arbitrarily distributed, the Array dimension is 1. For a 3D Grid with two arbitrarily distributed dimensions and one undistributed

dimension, the Array dimension is 2. In general, if the Array does not have any ungridded dimension, the Array dimension should be 1 plus the number of undistributed dimensions of the Grid.

The following is an example of creating an Array for a 3D Grid with 2 arbitrarily distributed dimensions such as the one defined in Section 31.3.7.

```
! Get distGrid from Grid
call ESMF_GridGet(grid3D, distgrid=distgrid, rc=rc)



! construct ArraySpec
call ESMF_ArraySpecSet(arrayspec, rank=2, typekind=ESMF_TYPEKIND_R8, rc=rc)



! Create an Array based on the presence of distributed dimensions
array=ESMF_ArrayCreate(arrayspec=arrayspec,distgrid=distgrid, rc=rc)
```

31.3.24 Create more complex Grids using DistGrid

Besides the shortcut methods for creating a Grid object such as `ESMF_GridCreateNoPeriDim()`, there is a set of methods which give the user more control over the specifics of the grid. The following describes the more general interface, using DistGrid. The basic idea is to first create an ESMF DistGrid object describing the distribution and shape of the Grid, and then to employ that to either directly create the Grid or first create Arrays and then create the Grid from those. This method gives the user maximum control over the topology and distribution of the Grid. See the DistGrid documentation in Section **??** for an in-depth description of its interface and use.

As an example, the following call constructs a 10x20 Grid with a lower bound of (1,2).

```
! Create DistGrid
distgrid2D = ESMF_DistGridCreate(minIndex=(/1,2/), maxIndex=(/11,22/), &
        rc=rc)



! Create Grid
grid3D=ESMF_GridCreate(distGrid=distgrid2D, rc=rc)
```

To alter which dimensions are distributed, the `distgridToGridMap` argument can be used. The `distgridToGridMap` is used to set which dimensions of the Grid are mapped to the dimensions described by `maxIndex`. In other words, it describes how the dimensions of the underlying default DistGrid are mapped to the Grid. Each entry in `distgridToGridMap` contains the Grid dimension to which the corresponding DistGrid dimension should be mapped. The following example illustrates the creation of a Grid where the largest dimension is first. To accomplish this the two dimensions are swapped.

```
! Create DistGrid
```

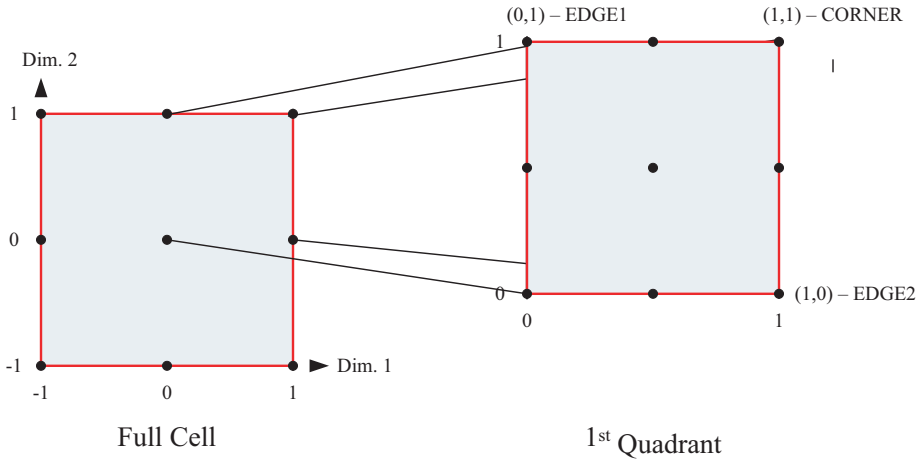


Figure 18: An example of specifying 2D stagger locations using coordinates.

```
distgrid2D = ESMF_DistGridCreate(minIndex=(/1,2/), maxIndex=(/11,22/), &
     rc=rc)



! Create Grid
grid2D=ESMF_GridCreate(distGrid=distgrid2D, distgridToGridMap=(/2,1/), &
     rc=rc)
```

31.3.25 Specify custom stagger locations

Although ESMF provides a set of predefined stagger locations (See Section 31.2.6), the user may need one outside this set. This section describes the construction of custom stagger locations.

To completely specify a stagger for an arbitrary number of dimensions, we define the stagger location in terms of a set of cartesian coordinates. The cell is represented by a n-dimensional cube with sides of length 2, and the coordinate origin located at the center of the cell. The geometry of the cell is for reference purposes only, and does not literally represent the actual shape of the cell. Think of this method instead as an easy way to specify a part (e.g. center, corner, face) of a higher dimensional cell which is extensible to any number of dimensions.

To illustrate this approach, consider a 2D cell. In 2 dimensions the cell is represented by a square. An xy axis is placed at its center, with the positive x-axis oriented *East* and the positive y-axis oriented *North*. The resulting coordinate for the lower left corner is at $(-1, -1)$, and upper right corner at $(1, 1)$. However, because our staggers are symmetric they don't need to distinguish between the -1, and the 1, so we only need to concern ourselves with the first quadrant of this cell. We only need to use the 1, and the 0, and many of the cell locations collapse together (e.g. we only need to represent one corner). See figure 18 for an illustration of these concepts.

The cell center is represented by the coordinate pair $(0, 0)$ indicating the origin. The cell corner is $+1$ in each direction,

giving a coordinate pair of $(1, 1)$. The edges are each $+1$ in one dimension and 0 in the other indicating that they're even with the center in one dimension and offset in the other.

For three dimensions, the vertical component of the stagger location can be added by simply adding an additional coordinate. The three dimensional generalization of the cell center becomes $(0, 0, 0)$ and the cell corner becomes $(1, 1, 1)$. The rest of the 3D stagger locations are combinations of $+1$ offsets from the center.

To generalize this to d dimensions, to represent a d dimensional stagger location. A set of d 0 and 1 is used to specify for each dimension whether a stagger location is aligned with the cell center in that dimension (0), or offset by $+1$ in that dimension (1). Using this scheme we can represent any symmetric stagger location.

To construct a custom stagger location in ESMF the subroutine `ESMF_StaggerLocSet()` is used to specify, for each dimension, whether the stagger is located at the interior (0) or on the boundary (1) of the cell. This method allows users to construct stagger locations for which there is no predefined value. In this example, it's used to set the 4D center and 4D corner locations.

```
! Set Center
call ESMF_StaggerLocSet(staggerLoc,loc=(/0,0,0,0/),rc=rc)


call ESMF_GridAddCoord(grid4D, staggerLoc=staggerLoc, rc=rc)




! Set Corner
call ESMF_StaggerLocSet(staggerLoc,loc=(/1,1,1,1/),rc=rc)



call ESMF_GridAddCoord(grid4D, staggerLoc=staggerLoc, rc=rc)
```

31.3.26 Specify custom stagger padding

There is an added complication with the data (e.g. coordinates) stored at stagger locations in that they can require different amounts of storage depending on the underlying Grid type.

Consider the example 2D grid in figure 19, where the dots represent the cell corners and the "+" represents the cell centers. For the corners to completely enclose the cell centers (symmetric stagger), the number of corners in each dimension needs to be one greater then the number of cell centers. In the above figure, there are two rows and three columns of cell centers. To enclose the cell centers, there must be three rows and four columns of cell corners. This is true in general for Grids without periodicity or other connections. In fact, for a symmetric stagger, given that the center location requires n x m storage, the corresponding corner location requires n+1 x m+1, and the edges, depending on the side, require n+1 x m or m+1 x n. In order to add the extra storage, a new DistGrid is created at each stagger location. This Distgrid is similar to the DistGrid used to create the Grid, but has an extra set of elements added to hold the index locations for the stagger padding. By default, when the coordinate arrays are created, one extra layer of padding is added to the index space to create symmetric staggers (i.e. the center location is surrounded). The default is to add this padding on the positive side, and to only add this padding where needed (e.g. no padding for the center, padding on both dimensions for the corner, in only one dimension for the edge in 2D.) There are two ways for the user to change these defaults.

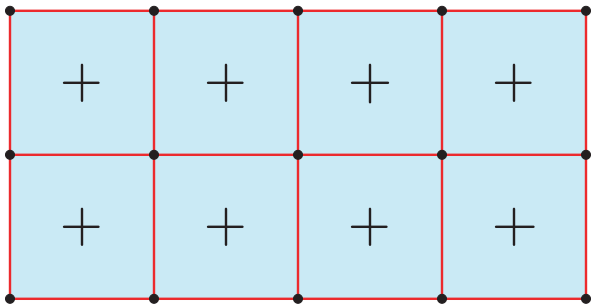


Figure 19: An example 2D Grid with cell centers and corners.

One way is to use the `GridEdgeWidth` or `GridAlign` arguments when creating a Grid. These arguments can be used to change the default padding around the Grid cell index space. This extra padding is used by default when setting the padding for a stagger location.

The `gridEdgeLWidth` and `gridEdgeUWidth` arguments are both 1D arrays of the same size as the Grid dimension. The entries in the arrays give the extra offset from the outer boundary of the grid cell index space. The following example shows the creation of a Grid with all the extra space to hold stagger padding on the negative side of a Grid. This is the reverse of the default behavior. The resulting Grid will have an exclusive region which extends from $(-1, -1)$ to $(10, 10)$, however, the cell center stagger location will still extend from $(1, 1)$ to $(10, 10)$.

```
grid2D=ESMF_GridCreateNoPeriDim(minIndex=(/1,1/),maxIndex=(/10,10/), &
         gridEdgeLWidth=(/1,1/), gridEdgeUWidth=(/0,0/), rc=rc)
```

To indicate how the data in a Grid's stagger locations are aligned with the cell centers, the optional `gridAlign` parameter may be used. This parameter indicates which stagger elements in a cell share the same index values as the cell center. For example, in a 2D cell, it would indicate which of the four corners has the same index value as the center. To set `gridAlign`, the values -1,+1 are used to indicate the alignment in each dimension. This parameter is mostly informational, however, if the `gridEdgeWidth` parameters are not set then its value determines where the default padding is placed. If not specified, then the default is to align all staggers to the most negative, so the padding is on the positive side. The following code illustrates creating a Grid aligned to the reverse of default (with everything to the positive side). This creates a Grid identical to that created in the previous example.

```
grid2D=ESMF_GridCreateNoPeriDim(minIndex=(/1,1/),maxIndex=(/10,10/), &
         gridAlign=(/1,1/), rc=rc)
```

The `gridEdgeWidth` and `gridAlign` arguments both allow the user to set the default padding to be used by stagger locations in a Grid. By default, stagger locations allocated in a Grid set their stagger padding based on these values. A stagger location's padding in each dimension is equal to the value of `gridEdgeWidth` (or the value implied by `gridAlign`), unless the stagger location is centered in a dimension in which case the stagger padding is 0. For example, the cell center stagger location has 0 stagger padding in all dimensions, whereas the edge stagger location lower padding is equal to `gridEdgeLWidth` and the upper padding is equal to `gridEdgeUWidth` in one