Earth System Modeling Framework

ESMF User Guide

Version 9.0.0 beta snapshot

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

November 25, 2026

---

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

1 What is the Earth System Modeling Framework? 5

2 The ESMF User's Guide 5

3 How to Contact User Support and Find Additional Information 6

4 How to Submit Comments, Bug Reports, and Feature Requests 6

5 Quick Start 7
- 5.1 Downloading ESMF . . . 7
- 5.2 Directory Structure . . . 7
- 5.3 Building ESMF . . . 7
  - 5.3.1 Environment variables . . . 8
  - 5.3.2 GNU make . . . 9
  - 5.3.3 make info . . . 9
  - 5.3.4 Building makefile targets . . . 13
  - 5.3.5 Testing makefile targets . . . 13
  - 5.3.6 Building and using bundled ESMF Command Line Tools . . . 14
- 5.4 Building ESMF with Spack . . . 15
  - 5.4.1 Creating New Spack environment . . . 15
  - 5.4.2 Finding Available Compilers and External Packages . . . 15
  - 5.4.3 Installing ESMF Spack Package and Its Dependencies . . . 15

6 Compiling and Linking User Code against an ESMF Installation 17
- 6.1 CMake method using config mode . . . 17
- 6.2 CMake method using module mode . . . 18
- 6.3 GNU Make method using the esmf.mk file . . . 18

7 Debugging of ESMF User Applications 21

8 Using Bundled ESMF Command Line Tools 24

9 Building and Installing ESMF 25
- 9.1 ESMF Download Options . . . 25
- 9.2 Acquiring Development Snapshots . . . 25
- 9.3 System Specific Information . . . 26
  - 9.3.1 General Requirements . . . 26
  - 9.3.2 Intel Compiler (Classic and LLVM-based) . . . 27
  - 9.3.3 MacOS Darwin . . . 28
- 9.4 Third Party Libraries . . . 28
  - 9.4.1 LAPACK . . . 28
  - 9.4.2 NetCDF . . . 29
  - 9.4.3 Parallel-NetCDF . . . 30
  - 9.4.4 PIO . . . 31
  - 9.4.5 Accelerator Software Stacks . . . 31
  - 9.4.6 XERCES . . . 32
  - 9.4.7 yaml-cpp . . . 33
  - 9.4.8 MOAB . . . 33
  - 9.4.9 NUMA . . . 34
  - 9.4.10 NVML . . . 34

9.5 ESMF Environment Variables . . . 35
9.6 Supported Platforms . . . 42
9.7 Building the ESMF Library . . . 44
9.8 Building the ESMF Documentation . . . 45
9.9 Installing the ESMF . . . 45

10 Porting ESMF **46**
10.1 The ESMF Build System . . . 46
10.1.1 General structure . . . 46
10.1.2 Build configuration . . . 47
10.1.3 Source code configuration . . . 48
10.2 Porting ESMF to New Platforms . . . 48
10.2.1 Customizing the `build_rules.mk` fragment . . . 48
10.2.2 Customizing `ESMC_Conf.h` and `ESMF_Conf.inc` . . . 53
10.3 Shared Object Libraries . . . 53

11 Validating an ESMF Build **54**
11.1 Running ESMF Self-Tests . . . 54
11.1.1 Setting up ESMF to run test suite applications . . . 54
11.1.2 Running ESMF unit tests . . . 56
11.1.3 Running ESMF system tests . . . 58
11.2 Running ESMF Examples . . . 60
11.2.1 Example source code . . . 60
11.2.2 Building and running examples . . . 60
11.3 Validating an existing ESMF installation . . . 62

12 Architectural Overview **64**
12.1 Key Concepts . . . 64
12.1.1 Modularity . . . 64
12.1.2 Flexibility . . . 64
12.1.3 Hierarchical organization . . . 65
12.1.4 Communication within Components . . . 65
12.1.5 Uniform communication API . . . 65
12.2 Superstructure . . . 65
12.2.1 Import and export State classes . . . 65
12.2.2 Interface standards . . . 67
12.2.3 Gridded Component class . . . 67
12.2.4 Coupler Component class . . . 67
12.2.5 Flexible data and control flow . . . 67
12.3 Infrastructure . . . 69
12.3.1 FieldBundle, Field and Array classes . . . 69
12.3.2 Grid class . . . 70
12.3.3 Time and Calendar management . . . 70
12.3.4 Config resource file manager . . . 70
12.3.5 DELayout and virtual machine . . . 70
12.3.6 Logging and error handling . . . 70
12.3.7 File input and output . . . 70

13 How to Adapt Applications for ESMF **70**
13.1 Individual Components . . . 71
13.2 Full Application . . . 72

14 Glossary **73**

References **79**

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.

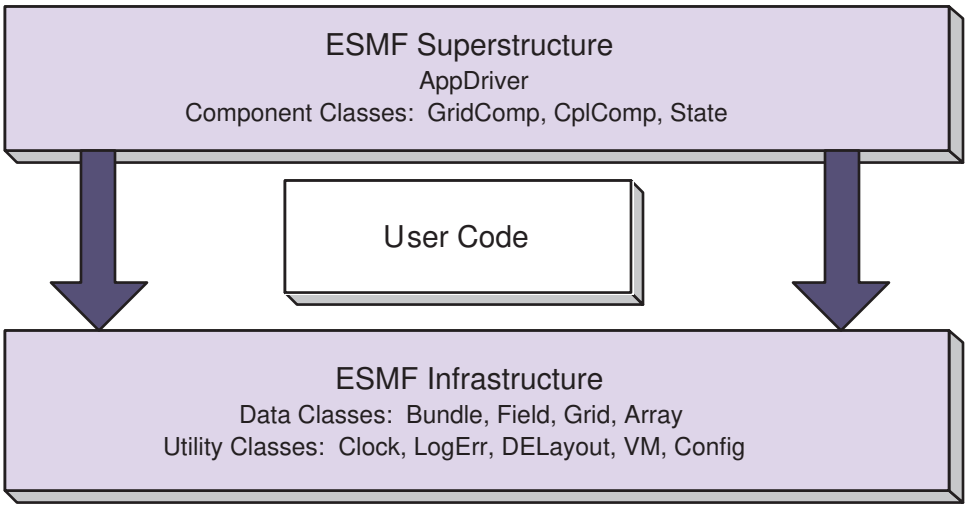
ESMF users may choose to extensively rewrite their codes to take advantage of the ESMF infrastructure, or they may decide to simply wrap their components in the ESMF superstructure in order to utilize framework coupling services. Either way, we encourage users to contact our support team if questions 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 User's Guide

This *ESMF User's Guide* is mainly an installation and build guide for the new ESMF user and a build reference for the experienced user. New users are strongly encouraged to download the ESMF software and try running the system tests and examples that illustrate both ESMF utilities and coupling services.

The *User's Guide* is organized as follows. The next two sections, 3 and 4, concern user support and how to submit comments on the ESMF system to our development team. Sections 5 through 11 contain a *Quick Start* guide that explains how to install the ESMF software and run the self-tests, followed by more detail on ESMF structure and

Figure 1: Schematic of the ESMF "sandwich" architecture. In this design 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.



operation, such as a description of the directory structure and how to build and run the ESMF example programs. Section 12 is an architectural overview that describes the framework's basic goals and features. Section 13 details the steps required to adapt a component for use with ESMF. Finally, to help you become familiar with ESMF terminology, the last section in the *User's Guide* is a glossary.

3 How to Contact User Support and Find Additional Information

The ESMF team can provide assistance in using the framework in your applications. For user support, please contact esmf_support@ucar.edu.

More information on the ESMF project as a whole is available on the ESMF website, http://www.earthsystemmodeling.org. 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 much more. Those curious about specific interfaces should refer to the *ESMF Reference Manual for Fortran*, which contains a detailed listing and description of the ESMF API (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 our project 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 Quick Start

This section gives a brief description of how to get the ESMF software, build it, and run the self-tests to verify the installation was successful. There is also a short guide for using the bundled ESMF command line tools. More detailed information on each of these steps is provided in sections 9, 11 and 8, respectively.

With a growing user community requiring access to ESMF, central computing resources have started providing system wide ESMF installations. The availablity of center-managed ESMF installations dramatically increases the ease of use of ESMF. Practically it means that if you are working on a system (such as *Jaguar*) that offers a standard ESMF installation, you do not have to download, build and validate your own ESMF installation from source! Instead you can proceed directly to using ESMF as a programming library or through access to the bundled command line tools as described in sections 6 and 8, respectively.

5.1 Downloading ESMF

ESMF is distributed via releases on GitHub. Each release page contains release notes, known issues, and links to supported platforms, documentation, and other related information Releases on GitHub can be found from the ESMF web page via:

```
http://www.earthsystemmodeling.org -> Download
```

5.2 Directory Structure

The current list of directories includes the following:

- README
- build
- build_config
- makefile
- scripts
- src

The `build_config` directory contains subdirectories for different operating system and compiler combinations. This is a useful area to examine if porting ESMF to a new platform.

5.3 Building ESMF

After downloading and unpacking the ESMF tar file, the build procedure is:

1. Set the required environment variables.
2. Type `make info` to view and verify your settings
3. Type `make` to build the library.

4. Type `make check` to run self-tests to verify the build was successful.

See the following subsections for more information on each of these steps. Also consult section 9 for a complete discussion of the the ESMF build process.

5.3.1 Environment variables

The syntax for setting environment variables depends on which shell you are running. Examples of the two most common ways to set an environment variable are:

ksh `export ESMF_DIR=/home/joeuser/esmf`

csh `setenv ESMF_DIR /home/joeuser/esmf`

The shell environment variables listed below are the ones most frequently used. There are others which address needs on specific platforms or are needed under more unusual circumstances; see section 9 for the full list.

ESMF_DIR The environment variable `ESMF_DIR` must be set to the full pathname of the top level ESMF directory before building the framework. This is the only environment variable which is required to be set on all platforms under all conditions.

ESMF_BOPT This environment variable controls the build option. To make a debuggable version of the library set `ESMF_BOPT` to `g` before building. The default is `O` (capital oh) which builds an optimized version of the library. If `ESMF_BOPT` is `O`, `ESMF_OPTLEVEL` can also be set to a numeric value between 0 and 4 to select a specific optimization level.

ESMF_COMM On systems with a vendor-supplied MPI communications library, the vendor library is chosen by default for communications. On these systems `ESMF_COMM` is set to `mpi`, signaling to the ESMF build system to use the vendor MPI implementation. For other systems (e.g. Linux or Darwin) where a multitude of MPI implementations are available, `ESMF_COMM` must be set to indicate which implementation is used to build the ESMF library. Set `ESMF_COMM` according to your situation to: `mpt`, `mpich (version 3 and up)`, `mpich1`, `mpich2`, `mvapich (all versions)`, `lam`, `openmpi`, or `intelmpi`. `ESMF_COMM` may also be set to `user` indicating that the user will set all the required flags using advanced ESMF environment variables. Some individual MPI builds may create additional libraries that need to be linked in, such as the legacy C++ bindings. These may be specified via the `ESMF_CXXLINKLIBS` and `ESMF_F90LINKLIBS` environment variables.

Alternatively, ESMF comes with a single-processor MPI-bypass library which is the default for Linux and Darwin systems. To force the use of this bypass library set `ESMF_COMM` equal to `mpiuni`.

ESMF_COMPILER The ESMF library build requires a working Fortran90 and C++ compiler. On platforms that don't come with a single vendor supplied compiler suite (e.g. Linux or Darwin) `ESMF_COMPILER` must be set to select which Fortran and C++ compilers are being used to build the ESMF library. Notice that setting the `ESMF_COMPILER` variable does *not* affect how the compiler executables are located on the system. `ESMF_COMPILER` (together with `ESMF_COMM`) affect the name that is expected for the compiler executables. Furthermore, the `ESMF_COMPILER` setting is used to select compiler and linker flags consistent with the compilers indicated.

By default Fortran and C++ compiler executables are expected to be located in a location contained in the user's `PATH` environment variable. This means that if you cannot locate the correct compiler executable via the `which` command on the shell prompt the ESMF build system won't find it either!

There are advanced ESMF environment variables that can be used to select specific compiler executables by specifying the full path. This can be used to pick specific compiler executables without having to modify the PATH environment variable.

Use 'make info' to see which compiler executables the ESMF build system will be using according to your environment variable settings.

To see possible values for ESMF_COMPILER, cd to $ESMF_DIR/build_config and list the directories there. The first part of each directory name corresponds to the output of 'uname -s' for this platform. The second part contains possible values for ESMF_COMPILER. In some cases multiple combinations of Fortran and C++ compilers are possible, e.g. there is intel and intelgcc available for Linux. Setting ESMF_COMPILER to intel indicates that both Intel Fortran and C++ compilers are used, whereas intelgcc indicates that the Intel Fortran compiler is used in combination with GCC's C++ compiler.

If you do not find a configuration that matches your situation you will need to port ESMF.

ESMF_ABI If a system supports 32-bit and 64-bit (pointer wordsize) application binary interfaces (ABIs), this variable can be set to select which ABI to use. Valid values are 32 or 64. By default the most common ABI is chosen. On x86_64 architectures three additional, more specific ABI settings are available, x86_64_32, x86_64_small and x86_64_medium.

ESMF_SITE Build configure file site name or the value default. If not set, then the value of default is assumed. When including platform-specific files, this value is used as the third part of the directory name (parts 1 and 2 are the ESMF_OS value and ESMF_COMPILER value, respectively.)

ESMF_ETCDIR If a user wants to add Attribute package specification files for their own customized Attribute packages, this is where they should go. ESMF will look in this directory for files that specify which Attributes are in an Attribute package for certain ESMF objects, and what the appropriate initial values would be for those Attributes. The format for these Attribute package specification files is to be defined in a future ESMF release. This environment variable is largely for internal use at this point.

ESMF_INSTALL_PREFIX This variable specifies the prefix of the installation path used during the installation process accessible thought the install target. Libraries, F90 module files, header files and documentation all are installed relative to ESMF_INSTALL_PREFIX by default. The ESMF_INSTALL_PREFIX may be provided as absolute path or relative to ESMF_DIR.

5.3.2 GNU make

GNU Make is required to build the ESMF library. On some systems this will be just the command make. On others it might be installed as gmake or gnumake. This document uses make consistently to refer to GNU Make.

Use the --version option with the locally available make commands to determine which variant corresponds to GNU Make on your system. Use the respective command when interacting with the ESMF build system, and where this documentation uses make.

Notice that ESMF does not utilize Autotools (configure or autoconf) or CMake. Instead, the selection of configuration options is done by setting environment variables before building the framework. The relevant environment variables all begin with prefix ESMF_, and are discussed in detail under section 9.5.

5.3.3 make info

make info is a command that assists the user in verifying that the ESMF variables have been set appropriately. It also tells the user the paths to various libraries e.g. MPI that are set on the system. The user to review this information

to verify their settings. In the case of a build failure, this information is invaluable and will be the first thing asked for by the ESMF support team. Below is an **example output** from `make info`:

```
------------------------------------------------------------
Make version:
GNU Make 3.80
Copyright (C) 2002  Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.

------------------------------------------------------------
Fortran Compiler version:
Intel(R) Fortran Compiler for applications running on Intel(R) 64, \
Version 10.1
Build 20081024 Package ID: l_fc_p_10.1.021
Copyright (C) 1985-2008 Intel Corporation.  All rights reserved.

Version 10.1

------------------------------------------------------------
C++ Compiler version:
Intel(R) C++ Compiler for applications running on Intel(R) 64, Version 10.1
Build 20081024 Package ID: l_cc_p_10.1.021
Copyright (C) 1985-2008 Intel Corporation.  All rights reserved.

Version 10.1

------------------------------------------------------------
Preprocessor version:
gcc (GCC) 4.1.2 20070115 (SUSE Linux)
Copyright (C) 2006 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.


------------------------------------------------------------
ESMF_VERSION_STRING:    5.1.0
------------------------------------------------------------

------------------------------------------------------------
 * User set ESMF environment variables *
ESMF_OS=Linux
ESMF_TESTMPMD=ON
ESMF_TESTHARNESS_ARRAY=RUN_ESMF_TestHarnessArrayUNI_2
ESMF_DIR=/nobackupp10/scvasque/daily_builds/intel/esmf
ESMF_TESTHARNESS_FIELD=RUN_ESMF_TestHarnessFieldUNI_1
ESMF_TESTWITHTHREADS=OFF
ESMF_COMM=mpiuni
ESMF_INSTALL_PREFIX= /nobackupp10/scvasque/daily_builds/intel/esmf/.. \
```

```
/install_dir
ESMF_TESTEXHAUSTIVE=ON
ESMF_BOPT=g
ESMF_SITE=default
ESMF_ABI=64
ESMF_COMPILER=intel

------------------------------------------------------------
 * ESMF environment variables *
ESMF_DIR: /nobackupp10/scvasque/daily_builds/intel/esmf
ESMF_OS:                Linux
ESMF_MACHINE:           x86_64
ESMF_ABI:               64
ESMF_COMPILER:          intel
ESMF_BOPT:              g
ESMF_COMM:              mpiuni
ESMF_SITE:              default
ESMF_PTHREADS:          ON
ESMF_OPENMP:            ON
ESMF_ARRAY_LITE:        FALSE
ESMF_NO_INTEGER_1_BYTE: FALSE
ESMF_NO_INTEGER_2_BYTE: FALSE
ESMF_FORTRANSYMBOLS:    default
ESMF_DEFER_LIB_BUILD:   ON
ESMF_TESTEXHAUSTIVE:    ON
ESMF_TESTWITHTHREADS:   OFF
ESMF_TESTMPMD:          ON
ESMF_TESTSHAREDOBJ:     OFF
ESMF_TESTFORCEOPENMP:   OFF
ESMF_TESTHARNESS_ARRAY: RUN_ESMF_TestHarnessArrayUNI_2
ESMF_TESTHARNESS_FIELD: RUN_ESMF_TestHarnessFieldUNI_1
ESMF_MPIRUN:            /nobackupp10/scvasque/daily_builds/intel/esmf/src/ \
                         Infrastructure/stubs/mpiuni/mpirun

------------------------------------------------------------
 * ESMF environment variables pointing to 3rd party software *

------------------------------------------------------------
 * ESMF environment variables for final installation *
ESMF_INSTALL_PREFIX:    /nobackupp10/scvasque/daily_builds/intel/esmf/../ \
install_dir
ESMF_INSTALL_HEADERDIR: include
ESMF_INSTALL_MODDIR:    mod/modg/Linux.intel.64.mpiuni.default
ESMF_INSTALL_LIBDIR:    lib/libg/Linux.intel.64.mpiuni.default
ESMF_INSTALL_BINDIR:    bin/bing/Linux.intel.64.mpiuni.default
ESMF_INSTALL_DOCDIR:    doc
ESMF_INSTALL_CMAKEDIR:  cmake


------------------------------------------------------------
 * Compilers, Linkers, Flags, and Libraries *
```

```
Location of the preprocessor:      /usr/bin/gcc
Location of the Fortran compiler:  /nasa/intel/fce/10.1.021/bin/ifort
Location of the Fortran linker:    /nasa/intel/fce/10.1.021/bin/ifort
Location of the C++ compiler:      /nasa/intel/cce/10.1.021/bin/icpc
Location of the C++ linker:        /nasa/intel/cce/10.1.021/bin/icpc

Fortran compiler flags:
ESMF_F90COMPILEOPTS: -g -fPIC -m64 -mcmodel=small -threads  -openmp
ESMF_F90COMPILEPATHS: -I/nobackupp10/scvasque/daily_builds/intel/esmf/mod/ \
modg/Linux.intel.64.mpiuni.default -I/nobackupp10/scvasque/daily_builds \
/intel/esmf/src/include
ESMF_F90COMPILECPPFLAGS: -DESMF_TESTEXHAUSTIVE -DSx86_64_small=1 \
-DESMF_OS_Linux=1  -DESMF_MPIUNI
ESMF_F90COMPILEFREECPP:
ESMF_F90COMPILEFREENOCPP:
ESMF_F90COMPILEFIXCPP:
ESMF_F90COMPILEFIXNOCPP:

Fortran linker flags:
ESMF_F90LINKOPTS:  -m64 -mcmodel=small -threads  -openmp
ESMF_F90LINKPATHS: -L/nobackupp10/scvasque/daily_builds/intel/esmf/lib/libg/ \
Linux.intel.64.mpiuni.default  -L/nasa/sgi/mpt/1.25/lib -L/nasa/intel/ \
cce/10.1.021/lib/shared -L/nasa/intel/fce/10.1.021/lib/shared -L/nasa/ \
intel/cce/10.1.021/lib -L/nasa/intel/fce/10.1.021/lib -L/nasa/intel/cce/ \
10.1.021/lib -L/usr/lib64/gcc/x86_64-suse-linux/4.1.2/ -L/usr/lib64/gcc/ \
x86_64-suse-linux/4.1.2/../../../../lib64
ESMF_F90LINKRPATHS:
-Wl,-rpath,/nobackupp10/scvasque/daily_builds/intel/esmf/lib/libg/ \
Linux.intel.64.mpiuni.default
ESMF_F90LINKLIBS:  -limf -lsvml -lm -lipgo -lguide -lstdc++ -lirc -lgcc_s \
-lgcc -lirc -lpthread -lgcc_s -lgcc -lirc_s -ldl -lrt -ldl
ESMF_F90ESMFLINKLIBS: -lesmf  -limf -lsvml -lm -lipgo -lguide -lstdc++ -lirc \
-lgcc_s -lgcc -lirc -lpthread -lgcc_s -lgcc -lirc_s -ldl -lrt -ldl

C++ compiler flags:
ESMF_CXXCOMPILEOPTS: -g -fPIC -m64 -mcmodel=small -pthread  -openmp
ESMF_CXXCOMPILEPATHS: -I/nobackupp10/scvasque/daily_builds/intel/ esmf/src/ \
include  -I/nobackupp10/scvasque/daily_builds/intel/esmf/src/Infrastructure \
/stubs/mpiuni
ESMF_CXXCOMPILECPPFLAGS: -DESMF_TESTEXHAUSTIVE  -DSx86_64_small=1 \
-DESMF_OS_Linux=1 -D__SDIR__='' -DESMF_MPIUNI

C++ linker flags:
ESMF_CXXLINKOPTS:  -m64 -mcmodel=small -pthread  -openmp
ESMF_CXXLINKPATHS: -L/nobackupp10/scvasque/daily_builds/intel/esmf/lib/libg/ \
Linux.intel.64.mpiuni.default  -L/nasa/intel/fce/10.1.021/lib/
ESMF_CXXLINKRPATHS: -Wl,-rpath,/nobackupp10/scvasque/daily_builds/intel/esmf/ \
lib/libg/Linux.intel.64.mpiuni.default -Wl,-rpath,/nasa/intel/fce/ \
10.1.021/lib/
ESMF_CXXLINKLIBS:  -lifport -lifcoremt -limf -lsvml -lm -lipgo -lguide -lirc \
-lpthread -lgcc_s -lgcc -lirc_s -ldl -lrt -ldl
```

```
ESMF_CXXESMFLINKLIBS: -lesmf  -lifport -lifcoremt -limf -lsvml -lm -lipgo \
-lguide -lirc -lpthread -lgcc_s -lgcc -lirc_s -ldl -lrt -ldl


------------------------------------------------------------
Compiling on Thu Oct 21 02:15:56 PDT 2010 on r75i0n8
Machine characteristics: Linux r75i0n8 2.6.16.60-0.68.1.20100916-nasa \
#1 SMP Fri
Sep 17 17:49:05 UTC 2010 x86_64 x86_64 x86_64 GNU/Linux
============================================================
```

5.3.4 Building makefile targets

The makefiles follow the GNU target standards where possible. The most frequently used targets for building are listed below:

lib build the ESMF libraries only (default)

all build the libraries, unit and system tests and examples

doc build the documentation (requires specific latex macros packages and additional utilities; see Section 9 for more details on the requirements).

info print out extensive system configuration information about what compilers, libraries, paths, flags, etc are being used

clean remove all files built for this platform/compiler/wordsize.

clobber remove all files built for all architectures

install install the ESMF library in a custom location

5.3.5 Testing makefile targets

To build and run the unit and system tests, type:

```
make check
```

A summary report of success and failures will be printed out at the end.

See section 11.1.1 on how to set up ESMF to be able to launch the bundled test and example applications.

Other test-related targets are:

all_tests build and run all available tests and examples

build_all_tests build tests and examples; do not execute

run_all_tests run tests and examples without rebuilding; print a summary of the results

check_all_tests print out the results summary without re-executing

dust_all_tests remove all test and example output files

clean_all_tests remove all test and example executables and output files

For all the targets listed above, the string `all_tests` can be replaced with one of the strings listed below to select a specific type of test:

unit_tests unit tests exercise a single part of the system

system_tests system tests combine functions across the system

examples examples contain code illustrating a single type of function

For example, `make build_examples` recompiles the example programs but does not execute them. `make dust_unit_tests` removes all output files generated when executing the unit tests, but leaves the executables. `make clean_system_tests` removes all executables and files associated with the system tests.

For the unit tests only, there is an additional environment variable which affects how the tests are built:

ESMF_TESTEXHAUSTIVE If this variable is set to `ON` before compiling the unit tests, longer and more exhaustive unit tests will be run. Note that this is a compile-time and not run-time option.

5.3.6 Building and using bundled ESMF Command Line Tools

This section describes how the bundled ESMF command line tools can be built and used from inside the ESMF source tree. Notice that this is sort of a quick and dirty way of accessing the ESMF applications. It is supported as convenience to those users interested in quickly gaining access to the bundled ESMF command line tools, and do not mind the shortcomings of this approach. Users interested in maximum portability should instead follow the instructions provided in section 8.

To build the bundled ESMF command line tools, type:

```
make build_apps
```

This will build the command line tools and place the executables under the `$ESMF_DIR/apps` directory inside the ESMF source tree. The command line tools can be directly executed from within the `$ESMF_DIR/apps` directory following the system specific rules for execution. The details will depend on whether ESMF was built with or without MPI dependency. In the latter case the system specific rules for launching parallel applications must be followed. System specific execution details on this level are outside of ESMF's scope.

For most systems, the MPI version of the ESMF bundled command line tools can be executed by a command equivalent to:

```
mpirun -np X $(ESMF_DIR)/apps/..../<cli-name>
```

where `X` specifies the total number of PETs and `cli-name` is the name of the specific ESMF command line tool to be executed. The `....` in the path indicates the precise subdirectory structure under `./apps` which follows the standard ESMF pattern also used for the `./tests` and `./examples` subdirectories.

All bundled ESMF command line tool support the standard `'--help'` command line option that prints out information on its proper use. More detailed instructions of the individual tools are available in the "Command Line Tools" section of the *ESMF Reference Manual*.

5.4 Building ESMF with Spack

In addition to the manual installation, ESMF can be installed through use of the *Spack* package manager. Since there are several ways to install Spack packages, this section aims to demonstrate creating a new, isolated Spack environment to install the ESMF library.

For a detailed documentation of the Spack package manager see the *Spack documentation* page. The *ESMF Spack package* page provides ESMF package specific information.

5.4.1 Creating New Spack environment

The following set of commands can be used to clone the Spack package manager, create a new environment in the current directory, and to activate the environment for ESMF installation.

```
git clone -c feature.manyFiles=true --depth=2 https://github.com/spack/spack.git
. spack/share/spack/setup-env.sh
spack env create -d $PWD/envs/myesmf
spack env activate $PWD/envs/myesmf
```

5.4.2 Finding Available Compilers and External Packages

Once a new Spack envronment is created and activated, available compilers and external packages can be added to the newly created environment using following Spack commands.

```
spack compiler find
spack external find
```

This will include locally available compilers and external packages to the environment specific `spack.yaml` file. In general, these are useful commands for detecting a small set of commonly-used packages but for now this is generally limited to finding build-only dependencies. The environment specific `spack.yaml` file can also be edited manually to include missing compilers and external packages to the environment. More information can be found in the *Spack documentation*.

5.4.3 Installing ESMF Spack Package and Its Dependencies

Since the ESMF Spack package includes different options to install the package, the following example just demonstrates commonly used ESMF package configurations. A full list of variants that can be used to install the ESMF Spack package can be found in the *ESMF Spack package* documentation.

To install ESMF with the external PIO package option, use the following commands:

```
spack add esmf@develop+external-parallelio
spack concretize --force --reuse
spack install
```

This will install the ESMF development version from the `develop` branch. A specific version of ESMF can be specified by replacing `develop` with desired version information, e.g. `8.8.1`. The user can also directly use the `spack install` command with the package name and desired variants, skipping the concretize step.

To install a specific ESMF tag that is not included in the official list of available versions can be installed using the `@=` syntax. E.g. to install ESMF beta tag `8.9.0b10`, the version would be specified as `esmf@=8.9.0b10`.

More information about specifying specific compilers and passing argument to the Spack package manager build system is available under the *Spack documentation*.

6 Compiling and Linking User Code against an ESMF Installation

Building user applications against an ESMF installation requires that the compiler and linker be able to find the appropriate ESMF header, module, and library files. If this procedure has been documented by the installer of the ESMF library on your system then follow the directions provided.

In the absence of site-specific instructions, there are standard CMake and GNU Make methods that ESMF supports to build user code against an ESMF installation.

6.1 CMake method using config mode

Starting with ESMF version 9, a standard `ESMFConfig.cmake` file is generated when ESMF built and installed. CMake tries to locate this `ESMFConfig.cmake` file in standard locations. During this search, CMake respects paths provided by the `CMAKE_PREFIX_PATH` variable in the user's environment. Spack-loaded installations of ESMF automatically set `CMAKE_PREFIX_PATH`, making this the most convenient method of building CMake-based applications against ESMF. For ESMF installations not managed by Spack, explicitly add the ESMF installation root directory to the `CMAKE_PREFIX_PATH` environment variable for seamless integration with CMake.

The configuration file provides exported imported targets that transitively bundle include directories, linked libraries, and compiler definitions (such as OpenMP or MPI requirements). To consume ESMF targets, utilize the `find_package()` command in your project's `CMakeLists.txt` file:

```
cmake_minimum_required(VERSION 3.22)
project(MyESMFApplication LANGUAGES Fortran C)

# 1. Locate the ESMF Package
# Specify a minimum version or version constraints if desired
# Explicitly request config mode
find_package(ESMF 9.0.0 REQUIRED CONFIG)

# 2. Define your application target
add_executable(my_model main.F90 physics_mod.F90)

# 3. Link against ESMF Imported Targets
# This automatically manages include paths, RPATHs, and downstream dependencies
target_link_libraries(my_model PUBLIC ESMF::ESMF)
```

Depending on whether the use application code is written in Fortran or C/C++, the ESMF package defines separate targets to avoid linking mismatch issues. Target aliases are provided for compatibility.

Target	Alias	Language Focus	Description
`ESMF::ESMF`	`ESMF::ESMF_Fortran`	Fortran	Configured for Fortran libraries and compiler variables, OpenMP flags, and tracking for `MPI::MPI_Fortran` dependencies.
`ESMF::ESMC`	`ESMF::ESMF_C`	C/C++	Configured with native C/C++ compiler variables, OpenMP flags, and tracking for `MPI::MPI_C` dependencies.

6.2 CMake method using module mode

An ESMF CMake find file is located at: *<ESMF source or install directory>/cmake/FindESMF.cmake*

The find file will parse the *esmf.mk* file created following a successful ESMF build. (See 6.3 for a description of the *esmf.mk* file.) The CMake module sets *esmf.mk* variables as global CMake variables. When using the find file, `ESMFMKFILE` must be set to the filepath of *esmf.mk*. If this is NOT set, then `ESMF_FOUND` will always be `FALSE`. If `ESMFMKFILE` exists, then `ESMF_FOUND=TRUE` and all ESMF makefile variables will be set in the global scope. Optionally, set `ESMF_MKGLOBALS` to a string list to filter makefile variables. For example, to globally scope only `ESMF_LIBSDIR` and `ESMF_APPSDIR` variables, use this CMake command in *CMakeLists.txt*: `set(ESMF_MKGLOBALS "LIBSDIR" "APPSDIR")`

6.3 GNU Make method using the esmf.mk file

Every ESMF installation provides a file named `esmf.mk` that contains the information needed to build a user application gainst the installation. The location of the `esmf.mk` file should be documented by the party that installed ESMF on the system. We recommend that a single ESMF specific environment variable, `ESMFMKFILE`, be provided by the system that points to the `esmf.mk` file. See section 9.9 for the related discussion aimed at the person that installs ESMF on a system.

The information in `esmf.mk` is defined in form of variables. In fact, syntactically `esmf.mk` is a makefile fragment and can be imported by an application specific makefile via the `include` command. All the variables in `esmf.mk` start with the "`ESMF_`" prefix to prevent conflicts. The information in `esmf.mk` is fully specified and is not affected by any variables set in the user's environment.

The information defined in `esmf.mk` includes Fortran compiler and linker, as well as C++ compiler and linker. It further includes the recommended Fortran and C++ specific compiler and linker flags for building ESMF applications. One way of using the `esmf.mk` is to glean the necessary information from it. This information can then be used either directly on the command line when compiling a user application, or to hardwire the settings into the application specific build system. However, the recommended use of `esmf.mk` is to include this file in the application specific makefile directly via the `include` command.

The `Makefile` template below demonstrates how a user build system can be constructed to leverage the `esmf.mk` file. In practice, most user build systems will be more complex. However, this template does show that the added complexity introduced by using `esmf.mk` is minimal. Examples of how to use this build system in realistic user scenarios can be found in the external demos.

The advantages of using `esmf.mk`, over hard coding suitable compiler and linker flags into the user build system directly, are robustness and portability. Robustness is a consequence of the fact that everything defined in `esmf.mk` corresponds to the exact settings used during the ESMF library build (consistency) and during the ESMF test suite build. Using `esmf.mk` thus guarantees that the user application is build in the exact same manner as the ESMF test suite applications that undergo strict regression testing before every ESMF release. Portability means that a user build system, which uses `esmf.mk` in the way the template `Makefile` demonstrates, will function as expected on any system where ESMF was successfully installed and tested, without the need of modifying anything. Every `esmf.mk` is generated during a specific ESMF installation using the ESMF tested settings for the host platform.

```
##############################################################################
### Makefile template for user ESMF application, leveraging esmf.mk mechanism ##
##############################################################################

##############################################################################
```

```
### Finding and including esmf.mk ##############################################

# Note: This fully portable Makefile template depends on finding environment
#       variable "ESMFMKFILE" set to point to the appropriate "esmf.mk" file,
#       as is discussed in the User's Guide.
#       However, you can still use this Makefile template even if the person
#       that installed ESMF on your system did not provide for a mechanism to
#       automatically set the environment variable "ESMFMKFILE". In this case
#       either manually set "ESMFMKFILE" in your environment or hard code the
#       location of "esmf.mk" into the include statement below.
#       Notice that the latter approach has negative impact on portability.

ifneq ($(origin ESMFMKFILE), environment)
$(error Environment variable ESMFMKFILE was not set.)
endif

include $(ESMFMKFILE)

################################################################################
### Compiler and linker rules using ESMF_ variables supplied by esmf.mk ########

.SUFFIXES: .f90 .F90 .c .C

.f90:
$(ESMF_F90COMPILER) -c $(ESMF_F90COMPILEOPTS) $(ESMF_F90COMPILEPATHS) \
          $(ESMF_F90COMPILEFREENOCPP) $<
$(ESMF_F90LINKER) $(ESMF_F90LINKOPTS) $(ESMF_F90LINKPATHS) \
          $(ESMF_F90LINKRPATHS) -o $@ $*.o $(ESMF_F90ESMFLINKLIBS)

.F90:
$(ESMF_F90COMPILER) -c $(ESMF_F90COMPILEOPTS) $(ESMF_F90COMPILEPATHS) \
          $(ESMF_F90COMPILEFREECPP) $(ESMF_F90COMPILECPPFLAGS) $<
$(ESMF_F90LINKER) $(ESMF_F90LINKOPTS) $(ESMF_F90LINKPATHS) \
          $(ESMF_F90LINKRPATHS) -o $@ $*.o $(ESMF_F90ESMFLINKLIBS)

.c:
$(ESMF_CXXCOMPILER) -c $(ESMF_CXXCOMPILEOPTS) \
          $(ESMF_CXXCOMPILEPATHSLOCAL) $(ESMF_CXXCOMPILEPATHS) \
          $(ESMF_CXXCOMPILECPPFLAGS) $<
$(ESMF_CXXLINKER) $(ESMF_CXXLINKOPTS) $(ESMF_CXXLINKPATHS) \
          $(ESMF_CXXLINKRPATHS) -o $@ $*.o $(ESMF_CXXESMFLINKLIBS)

.C:
$(ESMF_CXXCOMPILER) -c $(ESMF_CXXCOMPILEOPTS) \
          $(ESMF_CXXCOMPILEPATHSLOCAL) $(ESMF_CXXCOMPILEPATHS) \
          $(ESMF_CXXCOMPILECPPFLAGS) $<
$(ESMF_CXXLINKER) $(ESMF_CXXLINKOPTS) $(ESMF_CXXLINKPATHS) \
          $(ESMF_CXXLINKRPATHS) -o $@ $*.o $(ESMF_CXXESMFLINKLIBS)

################################################################################
### Sample targets for user ESMF applications ##################################
```

```
all: esmf_UserApplication esmc_UserApplication

esmf_UserApplication:

esmc_UserApplication:

##############################################################################
```

Notice that the ESMF_F90LINKPATHS, ESMF_F90LINKRPATHS, ESMF_CXXLINKPATHS, and ESMF_CXXLINKRPATHS variables used in the linking targets might contain paths to the specific compiler version, MPI implementation, and 3rd party libraries (see section 9.4) used when building ESMF. The paths are explicitly included in order to simplify the process of writing an application build system that is consistent with the ESMF library that is used.

There are, however, situations where it is desirable to let the application decide what compiler version, MPI version, and/or 3rd party library verson (e.g. NetCDF) to use. To this end, esmf.mk defines an alternative set of variables: ESMF_F90ESMFLINKPATHS, ESMF_F90ESMFLINKRPATHS, ESMF_CXXESMFLINKPATHS, and ESMF_CXXESMFLINKRPATHS. These variables only encode the precise path to the ESMF library, and do not specify where to find the compiler, MPI, and/or 3rd party libraries. When using this alternative set of variables, it becomes the responsibility of the application build system to ensure the required libraries can be found by the linker, and are compatible with the ESMF installation.

7 Debugging of ESMF User Applications

Debugging failing applications is often a challenging task. Massive parallelism, issues with compute node access, and large data volumes (just to name a few typical HPC aspects) add to the difficulties. For coupled applications, built from many individual components and libraries, additional complexity is introduced by the many layers of software.

For applications utilizing ESMF, the ESMF library is one of those software layers. Due to the "framework" nature of ESMF, the situation can be more subtle than for "simple" libraries. This is because ESMF code is called from user code (as for "simple" libraries), as well as calling back into registered user code (the "framework" aspect of ESMF). The consequences of this fact relating to debugging of applications are discussed in this section.

One consequence of the "framework" nature is that ESMF code is executing between major portions of user code. For instance, when one ESMF component calls into another ESMF component, the control flow goes through the ESMF software layer. This provides ESMF with a chance to write messages into an application wide log file. In particular, for user code that has implemented standard return code handling, ESMF can log an error trace in the event of detecting an error condition. The ESMF Reference Manual discusses standard "Return Code Handling" under a section of the same name.

By default, the application wide ESMF log output is written to files that are named `PET<nnn>.ESMF_LogFile`, where `<nnn>` is the number of the persistent execution thread (PET) that is writing. Several characteristics of the default log can be changed during the call to `ESMF_Initialize()`. In order to take advantage of the ESMF log output, it is important to ensure that the `logkindflag` is set to `ESMF_LOGKIND_MULTI`, which is the default, or `ESMF_LOGKIND_MULTI_ON_ERROR`. The latter is recommended for production runs where extra log output is minimized, and the ESMF log is only activated when an error is encountered.

Assuming that the ESMF log is active for a failing application, and the user code follows the documented return code handling, the ESMF log files are among the first files that should be inspected. The log files are written into the working directory that was active during the application execution. Assuming default log file naming, we recommend the following grep command to scan for errors.

```
grep ERROR PET*.ESMF_LogFile
```

A typical error trace looks similar to the following output. Here is an example error trace for an application using the NUOPC layer.

```
20210317 150338.047 ERROR PET0 atm.F90:113       Invalid argument  - \
                                          Passing error in return code
20210317 150338.047 ERROR PET0 ATM:src/addon/NUOPC/src/NUOPC_ModelBase.F90:865
20210317 150338.047 ERROR PET0 esm:src/addon/NUOPC/src/NUOPC_Driver.F90:2570
20210317 150338.047 ERROR PET0 esm:src/addon/NUOPC/src/NUOPC_Driver.F90:1287
20210317 150338.047 ERROR PET0 esm:src/addon/NUOPC/src/NUOPC_Driver.F90:466
20210317 150338.047 ERROR PET0 esmApp.F90:64       Invalid argument  - \
                                           Passing error in return code
```

The first two columns contain the wall clock information of when each individual log message was written. The third and forth column indicate the type of the log message, here `ERROR`, and the PET number, respectively, The sixth column contains information about the source file and line number. Finally the seventh column and beyond contain information about the error.

Notice that error traces are logged in backward order. The first `ERROR` entry corresponds to the lowest level, where the error condition was first detected. Here the error was first detected in file `atm.F90`, at line 113. The error is then propagated back up to the highest application level, here ending up in file `esmApp.F90`, line 64. Due to the

framework nature of ESMF discussed earlier, it is very common to see several layers of ESMF library code in an error trace, as is the case in this example. Notice however, that the error was first caught in "user code" `atm.F90`.

Even if the lowest level indicated (i.e. the start of an error trace) is inside the ESMF library, it does not immediately indicate an issue with ESMF code. In such cases it is good to follow the error trace to the first user code entry, and investigate what ESMF call is made just before that location. Then consider looking at the specific information passed into the ESMF method and ensure correctness.

There are situations where an application experiences a hard crash, either triggered by the runtime library, or the operating system itself. In these cases the ESMF log files are typically not as helpful, and might even be missing. A hard crash that produces a code dump, a backtrace to stderr, or is caught under a debugger, can still be a good source of information to track down the problematic issue.

An example backtrace for a hard crash is shown below.

```
Program received signal SIGFPE: Floating-point exception - erroneous arithmetic operation.

Backtrace for this error:
#0  0x7f9e45bed49f in ???
#1  0x40430e in realize
        at /tmp/AtmOcnProto/ocn.F90:149
#2  0x7f9e49568f8b in _ZNK5ESMCI13MethodElement7executeEPvPi
        at /tmp/esmf/src/Superstructure/Component/src/ESMCI_MethodTable.C:333
#3  0x7f9e49569e74 in _ZN5ESMCI11MethodTable7executeENSt7__cxx1112basic_stringIcSt11char_t
        at /tmp/esmf/src/Superstructure/Component/src/ESMCI_MethodTable.C:519
#4  0x7f9e49568dea in c_esmc_methodtableexecuteef_
        at /tmp/esmf/src/Superstructure/Component/src/ESMCI_MethodTable.C:303
#5  0x7f9e4982d8da in __esmf_attachmethodsmod_MOD_esmf_methodgridcompexecute
        at /tmp/esmf/src/Superstructure/AttachMethods/src/ESMF_AttachMethods.F90:1278
#6  0x7f9e4a69fcd0 in initializeipdvxp04
        at /tmp/esmf/src/addon/NUOPC/src/NUOPC_ModelBase.F90:1263
#7  0x7f9e492b8d1a in _ZN5ESMCI6FTable12callVFuncPtrEPKcPNS_2VMEPi
        at /tmp/esmf/src/Superstructure/Component/src/ESMCI_FTable.C:2036
#8  0x7f9e492b605a in ESMCI_FTableCallEntryPointVMHop
        at /tmp/esmf/src/Superstructure/Component/src/ESMCI_FTable.C:765
#9  0x7f9e49654bc3 in _ZN5ESMCI3VMK5enterEPNS_7VMKPlanEPvS3_
        at /tmp/esmf/src/Infrastructure/VM/src/ESMCI_VMKernel.C:2195
#10  0x7f9e49668136 in _ZN5ESMCI2VM5enterEPNS_6VMPlanEPvS3_
        at /tmp/esmf/src/Infrastructure/VM/src/ESMCI_VM.C:1211
#11  0x7f9e492b64f0 in c_esmc_ftablecallentrypointvm_
        at /tmp/esmf/src/Superstructure/Component/src/ESMCI_FTable.C:922
#12  0x7f9e499f8678 in __esmf_compmod_MOD_esmf_compexecute
        at /tmp/esmf/src/Superstructure/Component/src/ESMF_Comp.F90:1216
#13  0x7f9e49e5e19b in __esmf_gridcompmod_MOD_esmf_gridcompinitialize
        at /tmp/esmf/src/Superstructure/Component/src/ESMF_GridComp.F90:1408
#14  0x7f9e4a63740e in loopmodelcompss
        at /tmp/esmf/src/addon/NUOPC/src/NUOPC_Driver.F90:2534
#15  0x7f9e4a641e92 in initializeipdv02p3
        at /tmp/esmf/src/addon/NUOPC/src/NUOPC_Driver.F90:1833
#16  0x7f9e4a6650c9 in initializep1
        at /tmp/esmf/src/addon/NUOPC/src/NUOPC_Driver.F90:467
#17  0x7f9e492b8d1a in _ZN5ESMCI6FTable12callVFuncPtrEPKcPNS_2VMEPi
```

```
        at /tmp/esmf/src/Superstructure/Component/src/ESMCI_FTable.C:2036
#18  0x7f9e492b605a in ESMCI_FTableCallEntryPointVMHop
        at /tmp/esmf/src/Superstructure/Component/src/ESMCI_FTable.C:765
#19  0x7f9e49654bc3 in _ZN5ESMCI3VMK5enterEPNS_7VMKPlanEPvS3_
        at /tmp/esmf/src/Infrastructure/VM/src/ESMCI_VMKernel.C:2195
#20  0x7f9e49668136 in _ZN5ESMCI2VM5enterEPNS_6VMPlanEPvS3_
        at /tmp/esmf/src/Infrastructure/VM/src/ESMCI_VM.C:1211
#21  0x7f9e492b64f0 in c_esmc_ftablecallentrypointvm_
        at /tmp/esmf/src/Superstructure/Component/src/ESMCI_FTable.C:922
#22  0x7f9e499f8678 in __esmf_compmod_MOD_esmf_compexecute
        at /tmp/esmf/src/Superstructure/Component/src/ESMF_Comp.F90:1216
#23  0x7f9e49e5e19b in __esmf_gridcompmod_MOD_esmf_gridcompinitialize
        at /tmp/esmf/src/Superstructure/Component/src/ESMF_GridComp.F90:1408
#24  0x401aaf in esmapp
        at /tmp/AtmOcnProto/esmApp.F90:58
#25  0x401ee4 in main
        at /tmp/AtmOcnProto/esmApp.F90:17
Floating exception (core dumped)
```

Just as for the ESMF log, the backtrace is produced in reverse order, starting at the lowest level where the problem was encountered, tracing all the way up the call stack to `main`. As expected, the backtrace contains many layers of ESMF library code. Again this does not immediately indicate that there is a problem in the library code. In this example user code is visible at the very top of the stack `esmApp.F90`, and at the very bottom `ocn.F90`. In fact the location of the division by zero is correctly identified by the runtime library at line 149 in file `ocn.F90`.

It is possible to force a hard crash within the ESMF library while logging ERRORS or WARNINGS to the `PET<nnn>.ESMF_LogFile`. Doing so can be advantageous because it may produce a code dump and/or backtrace at the initial point of error without ESMF return code handling. There are two settings controlling log message error handling. The first setting, `ESMF_RUNTIME_ABORT_LOGMSG_TYPES`, configures the ESMF library to abort during specified log message types, such as `ESMF_LOGMSG_ERROR`. When setting `ESMF_RUNTIME_ABORT_LOGMSG_TYPES`, multiple log message types can be listed, such as `ESMF_LOGMSG_ERROR,ESMF_LOGMSG_WARNING`. The second setting, `ESMF_RUNTIME_ABORT_ACTION`, configures the ESMF library abort action. By default, the ESMF library abort handler will call `MPI_Abort`. This can be changed to `SIGABRT` or `SIGQUIT`, which will raise their respective exceptions. These two settings can be configured in the user environment before initializing ESMF or set in the configuration file passed to `ESMF_Initialize`.

An example configuration file configuring ESMF to raise `SIGABRT` during `ESMF_LOGMSG_ERROR` or `ESMF_LOGMSG_WARNING`.

```
ESMF_RUNTIME_ABORT_ACTION: "SIGABRT"
ESMF_RUNTIME_ABORT_LOGMSG_TYPES: "ESMF_LOGMSG_ERROR,ESMF_LOGMSG_WARNING"
```

8 Using Bundled ESMF Command Line Tools

ESMF comes with a set of bundled command line tools (CLT). These applications include convenient access to general information about an ESMF installation, and regrid weight file generation (sometimes referred to as "offline" regridding). This section provides assistance with respect to building and running the bundled CLTs. If you are using a pre-installed ESMF on your system, follow the local instructions provided by the installer or system admin of how to access and run the ESMF CLTs. Often access is as simple as loading a configuration module to have the correct path to the ESMF CLT binaries added to your `PATH` environment variable.

There are two ways a user may choose to build and access the bundled ESMF CLTs. Users that prefer not to go through the full ESMF installation process have the option to build the bundled CLTs inside of the ESMF source tree, very similar to how the unit tests, system tests and examples are built. This option is outlined in section 5.3.6 and should only be considered by users that want quick access to the CLTs and are not interested in a sharable installation or the development of portable scripts and makefiles that use the CLTs. Users interested in the latter should consider the more standard second option outlined below.

The bundled ESMF CLTs are built automatically in the process of installing ESMF following the instructions given in section 9.9. On systems that offer system-wide ESMF installations (e.g. via modules or similar mechanisms) the user need not worry about the build and installation details. Once installed, the CLTs are accessible through their precise location on the system. For this purpose every ESMF installation provides a file named `esmf.mk` that contains the variable `ESMF_APPSDIR` which specifies the precise CLT path.

The `esmf.mk` mechanism used for CLT access is the same as the one described in section 6 for writing robust and portable user makefiles for building and linking user CLTs against an ESMF installation. One feature of the `esmf.mk` mechanism is that only one single piece of information must be known about an ESMF installation to use it, and that is the location of file `esmf.mk` itself. The location of this file should be documented by the party that installed ESMF on the system. We recommend that a single ESMF specific environment variable ESMFMKFILE be provided by the system that points to the `esmf.mk` file. See section 9.9 for the related discussion aimed at the person that installs ESMF on a system.

Once the exact location of the bundled ESMF CLT files has been determined, either by inspecting the associated `esmf.mk` file, or by using the `ESMF_APPSDIR` makefile variable directly in the user script or makefile, the CLTs can be executed following the system specific rules for execution. The details will depend on whether ESMF was built with or without MPI dependency. In the latter case the system specific rules for launching parallel CLTs must be followed. System specific execution details on this level are outside of ESMF's scope. However, ESMF does offer specific CLT use examples as part of the *external_demos* module described online at the External Demos webpage. For most systems, the MPI version of the ESMF bundled CLTs can be executed by a command equivalent to:

```
mpirun -np X $(ESMF_APPSDIR)/<clt-name>
```

where `X` specifies the total number of PETs and `clt-name` is the name of the specific ESMF command line tool to be executed.

All bundled ESMF CLTs support the standard `'--help'` command line option that prints out information on its proper use. More detailed instructions of the individual CLTs are available in the "Command Line Tools" section of the *ESMF Reference Manual*.

9 Building and Installing ESMF

This section goes into more detail about how to build and install the ESMF software.

9.1 ESMF Download Options

Major releases of the ESMF software can be downloaded by following the instructions on the the **Download** link on the ESMF website, http://www.earthsystemmodeling.org.

The ESMF is distributed as a full source code tree. Follow the instructions in the following sections to build the library and link it with your application.

9.2 Acquiring Development Snapshots

Occasionally, it is helpful to acquire a development snapshot of ESMF in order to test emerging capabilities, optimizations, and bug fixes before they are available in a formal release. Development snaphots are "use at your own risk." Efforts are made to ensure that most unit and system tests are passing on typical platforms, but there are no guarantees of the stability of development snapshots. New APIs available in development snapshots may change before the next release.

Users aware of these risks may check out development snapshots using the appropriate git tag.

Starting with ESMF 8.3.0 beta snapshot 07, the naming convention for development tags has the form:

```
v<VERSION>b<NUMBER>
```

For example:

```
v8.3.0b07
```

Prior to this version, the tag naming convention for development tags is:

```
ESMF_<VERSION>_beta_snapshot_<NUMBER>
```

For example:

```
ESMF_8_2_0_beta_snapshot_23
```

Use the following example command as a guide to check out a specific development tag:

```
  git clone https://github.com/esmf-org/esmf.git --branch v8.3.0b13 --depth 1
```

Once downloaded, development snapshots are built in the same way as releases.

9.3 System Specific Information

9.3.1 General Requirements

The following compilers and utilities are required for compiling, linking and testing the ESMF software. It is good common practice to use a consistent set of Fortran/C++/C compilers from the same vendor, e.g. GNU, Intel, etc. However, some vendor *combinations* of Fortran, C++, and C compilers, e.g. Intel ifort with GNU g++, are also supported.

- Fortran compiler:
  - GNU's gfortran v7.0 and newer, or
  - Intel's ifort v18.0 and newer, or
  - PGI's pgf90 v18.1 and newer, or
  - NVHPC's nvfortran, or
  - NAG's nagfor, or
  - IBM's xlf, or
  - CCE's ftn.
- C++ compiler:
  - GNU's g++ v7.0 and newer, or
  - Intel's icpc v18.0 and newer, or
  - PGI's pgCC v18.1 and newer, or
  - NVHPC's nvc++, or
  - IBM's xlC, or
  - CCE's CC.
  - LLVM's clang
- C compiler:
  - GNU's gcc v7.0 and newer, or
  - Intel's icc v18.0 and newer, or
  - PGI pgcc v18.1 and newer, or
  - NVHPC's nvcc, or
  - IBM's xlc, or
  - CCE's cc.
  - LLVM's clang
- MPI implementation compatible with the above compilers (but also see below for the MPI-bypass build option):
  - OpenMPI v3.0 and newer, or
  - MPICH v2.1 and newer, or
  - MVAPICH v2.0 and newer, or
  - IntelMPI v18.0 and newer, or
  - MPT 2.17 and newer, or

  - CRAY-MPICH v7.7 and newer.

- GNU's gcc compiler - for a standard cpp preprocessor implementation.
- GNU Make.
- Perl - for running test scripts.

Internal packages that can optionally reference external libraries:

- LAPACK - version 3.x or newer
- ParallelIO (PIO) - version 2.5.10 or newer
- yaml-cpp - tag yaml-cpp-0.6.2 or newer

Optional external packages that must be specified for certain functions:

- NetCDF - version 3.6.x or newer (version 4.4 or newer required by PIO)
- parallel-NetCDF - version 1.2.0 or newer (version 1.12 or newer required by PIO)
- Xerces - version 3.1.0 or newer

ESMF can be built using a single-processor MPI-bypass library that comes with ESMF by setting `ESMF_COMM=mpiuni`. This allows ESMF applications to be linked and run in single-process mode.

In order to build html and pdf versions of the ESMF documentation, LaTeX, the latex2html conversion utility, and the Unix/Linux `dvipdf` utility must be installed. The csh shell is also required to complete the documentation build.

9.3.2 Intel Compiler (Classic and LLVM-based)

ESMF supports the Intel compiler suite via `ESMF_COMPILER=intel`. Starting in 2020, Intel began promoting their new LLVM-based C/C++ and Fortran compiler line as part of the oneAPI brand. During a multi-year transition period, the oneAPI product shipped with both classic (`icc`, `icpc`, `ifort`) *and* LLVM-based (`icx`, `icpx`, `ifx`) compilers. As of version 2024.0 of oneAPI, only the LLVM-based compilers are provided by Intel. ESMF supports both Intel compiler lines. The following paragraphs provide important details that allow users to fine-tune the interaction with the Intel compiler suite.

Under `ESMF_OS=Linux`, with `ESMF_COMPILER=intel` and `ESMF_COMM=mpiuni` set, the C, C++, and Fortran compiler front-ends default to the LLVM-based option `icx`, `icpx`, and `ifx`, respectively. Any of these defaults can be overridden by explicitly setting the `ESMF_C`, `ESMF_CXX`, or `ESMF_F90` environment variables, e.g. to the classic compiler options `icc`, `icpc`, and `ifort`, respectively.

Under `ESMF_OS=Linux`, with `ESMF_COMPILER=intel` and `ESMF_COMM=intelmpi` set, the C, C++, and Fortran compiler front-ends default to the MPI compiler wrappers `mpiicc`, `mpicpc`, and `mpiifort`, respectively. It depends on the IntelMPI installation details whether classic, LLVM-based, or a mixture of compilers are used underneath the MPI wrappers. The IntelMPI defaults can be overridden by explicitly setting the IntelMPI-specific environment variables `I_MPI_CC`, `I_MPI_CXX`, or `I_MPI_F90`. This is an IntelMPI feature.

The recommendation for Cray systems that utilize the Cray Programming Environment (CPE) is to use `ESMF_OS=Unicos`. In most cases this setting is detected automatically by the ESMF build system, and should *not* be overridden.

Under ESMF_OS=Unicos, with ESMF_COMPILER=intel, the C, C++, and Fortran compiler front-ends default to cc, CC, and ftn, respectively, regardless of the ESMF_COMM setting. The appropriate classic, oneAPI, or mixed compiler combination is typically determined by the Intel environment module loaded. Common module names on Cray systems are intel-classic, intel-oneAPI, and intel, respectively.

Tip: Use the ESMF "make info" target to query detailed compiler version information. This can be used to determine the appropriate settings, and to diagnose issues before kicking off the complete ESMF build procedure.

9.3.3 MacOS Darwin

ESMF supports MacOS systems via the ESMF_OS=Darwin setting, which typically is auto-detected by the ESMF build system. Various compilers and MPI implementations are supported under Darwin using the ESMF_COMPILER and ESMF_COMM environment variables. There are some combinations under Darwin that require special attention; these combinations are listed below.

On Darwin with ESMF_COMPILER=gfortran and ESMF_COMM=mpich, using MPICH3 built from source, it is important to specify the -enable-two-level-namespace configure option when building the MPICH3 library. By default, i.e. without this option, the produced MPICH compiler wrappers include a linker flag (-flat_namespace) that causes issues with C++ exception handling under GNU g++. Building and linking ESMF applications with MPICH compiler wrappers that specify this linker option leads to "mysterious" application aborts during execution.

On Darwin with ESMF_COMPILER=intel, command line arguments cannot be accessed from ESMF applications when linked against the shared library version (libesmf.dylib). There is no issue when linked against the static version (libesmf.a). Setting the environment variable ESMF_SHARED_LIB_BUILD=OFF when building ESMF can be used as a work around for this issue.

9.4 Third Party Libraries

Some portions of the ESMF library can offer enhanced capabilities when certain third party libraries are available. This section describes these dependencies and the associated environment settings that allow the user to control them.

On many platforms, the ESMF library is also created as a shared library. When third party libraries are called from ESMF, it is recommended that they are also available as shared libraries. In cases where they are not, they should at least be compiled with the position independent code option enabled (e.g., -fPIC on Linux with gfortran/gcc) where necessary, so that the ESMF shared library build can successfully incorporate them.

9.4.1 LAPACK

The patch recovery regridding method of the ESMF Mesh class requires solving local least squares problems. It uses the LAPACK *DGELSY* solver to carry out this task.

The following environment variables control whether a minimal set of LAPACK code that comes with ESMF is used, or whether ESMF should link against an externally available LAPACK installation. Alternatively, ESMF's LAPACK-dependent features can be turned off altogether.

ESMF_LAPACK Possible value: "internal" (default), "OFF", "system", "mkl", "netlib", "scsl", openblas, <*userstring*>.

"internal" **(default)** ESMF will be compiled with LAPACK-dependent features. A minimal set of LAPACK/BLAS code included in ESMF will be used to satisfy the dependencies.

"OFF" Disables LAPACK-dependent code.

"system" A system-dependent external LAPACK/BLAS installation is used to satisfy the external dependencies of the LAPACK-dependent ESMF code. Sets ESMF_LAPACK_LIBS appropriately.

"mkl" The Intel MKL library is used to satisfy the external dependencies of the LAPACK-dependent ESMF code. When ESMF_COMPILER is set to "intel", ESMF_LAPACK_LIBS is set to "-mkl". Otherwise ESMF_LAPACK_LIBS is set to "-lmkl_lapack -lmkl", unless it is already defined in the user environment.

"netlib" The NETLIB library is used to satisfy the external dependencies of the LAPACK-dependent ESMF code. Sets ESMF_LAPACK_LIBS to "-llapack -lblas", unless it is already defined in the user environment.

"scsl" The SCSL library is used to satisfy the external dependencies of the LAPACK-dependent ESMF code. Sets ESMF_LAPACK_LIBS to "-lscs", unless it is already defined in the user environment.

"openblas" The OpenBLAS library is used to satisfy the external dependencies of the LAPACK-dependent ESMF code. Sets ESMF_LAPACK_LIBS to "-openblas", unless it is already defined in the user environment.

<*userstring*> Enables ESMF's LAPACK-dependent code, but does not set a default for ESMF_LAPACK_LIBS. ESMF_LAPACK_LIBS, and if required, ESMF_LAPACK_LIBPATH, must be set explicitly in the user environment.

ESMF_LAPACK_LIBPATH Typical value: /usr/local/lib (no default).

Specifies the path where the LAPACK library is located.

ESMF_LAPACK_LIBS Typical value: "-llapack -lblas" (default is dependent on ESMF_LAPACK).

Specifies the linker directive needed to link the LAPACK library to the application. On some systems, the BLAS library must also be included.

9.4.2 NetCDF

ESMF provides the ability to read Grid and Mesh data in NetCDF format.

Beginning with NetCDF 4.2, the C and Fortran API libraries are released as separate packages. To compile ESMF with NetCDF 4.2 and newer releases, the ESMF_NETCDF environment variable can be set to "split". The "split" option requires the NetCDF C library, and the NetCDF Fortran API library be installed in the same directory. As an alternative, the "nc-config" option may be used to automatically determine the include and lib directory locations. The "nc-config" option supports separate C and Fortran directories.

The following environment variables enable, and specify the name and location of the desired NetCDF library and associated header files:

ESMF_NETCDF Possible value: *not set* (default), "nc-config", "split", "standard", <*userstring*>.

not set **(default)** NetCDF-dependent features will be disabled. The ESMF_NETCDF_INCLUDE, ESMF_NETCDF_LIBPATH, and ESMF_NETCDF_LIBS environment variables will be ignored.

"nc-config" The NetCDF nc-config and if available, nf-config, tools will be used to determine the proper settings of ESMF_NETCDF_INCLUDE, ESMF_NETCDF_LIBPATH, and ESMF_NETCDF_LIBS. The shell PATH environment variable must include the NetCDF bin directories where nc-config and nf-config reside. This option supports having the main NetCDF library and the Fortran API library reside in separate directories.

"split" ESMF_NETCDF_LIBS will be set to "-lnetcdff -lnetcdf". This option is useful for systems which have the Fortran and C bindings archived in separate library files. The ESMF_NETCDF_INCLUDE and ESMF_NETCDF_LIBPATH environment variables will also be used, if defined.

"standard" ESMF_NETCDF_LIBS will be set to "-lnetcdf". This option is useful when the Fortran and C bindings are archived together in the same library file. The ESMF_NETCDF_INCLUDE and ESMF_NETCDF_LIBPATH environment variables will also be used, if defined.

<userstring> If set, ESMF_NETCDF_INCLUDE, ESMF_NETCDF_LIBPATH, and ESMF_NETCDF_LIBS environment variables will be used, if defined.

ESMF_NETCDF_INCLUDE Typical value: /usr/local/include (no default).

Specifies the path where the NetCDF header files are located.

ESMF_NETCDF_LIBPATH Typical value: /usr/local/lib (no default).

Specifies the path where the NetCDF library file is located.

ESMF_NETCDF_LIBS Typical value: "-lnetcdf"

Specifies the linker directives needed to link the NetCDF library to the application.

The default value depends on the setting of ESMF_NETCDF. For the typical case where ESMF_NETCDF is set to "standard", ESMF_NETCDF_LIBS is set to "-lnetcdf". When ESMF_NETCDF is set to "split", ESMF_NETCDF_LIBS is set to "-lnetcdff -lnetcdf".

If the hdf5 library is required, append "-lhdf5_hl -lhdf5" to the desired setting. E.g. "-lnetcdff -lnetcdf -lhdf5_hl -lhdf5"

9.4.3 Parallel-NetCDF

ESMF provides the ability to write data using Parallel-NetCDF.

Some file systems, for example Lustre, may need to have locking attributes enabled when the file system is mounted.

The following environment variables enable and specify the name and location of the desired Parallel-NetCDF library and associated header files:

ESMF_PNETCDF Possible value: *not set* (default), "pnetcdf-config", "standard", *<userstring>*.

When defined, enables the use of Parallel-NetCDF.

not set **(default)** PNETCDF-dependent features will be disabled. The ESMF_PNETCDF_INCLUDE, ESMF_PNETCDF_LIBPATH, and ESMF_PNETCDF_LIBS environment variables will be ignored.

"pnetcdf-config" The PNetCDF pnetcdf-config tool will be used to determine the proper settings of ESMF_PNETCDF_INCLUDE, ESMF_PNETCDF_LIBPATH, and ESMF_PNETCDF_LIBS. The shell PATH environment variable must include the PNetCDF bin directory where pnetcdf-config resides.

"standard" ESMF_PNETCDF_LIBS will be set to "-lpnetcdf". The ESMF_PNETCDF_INCLUDE and ESMF_PNETCDF_LIBPATH environment variables will also be used, if defined.

<userstring> If set, ESMF_PNETCDF_INCLUDE, ESMF_PNETCDF_LIBPATH, and ESMF_PNETCDF_LIBS environment variables will be used.

ESMF_PNETCDF_INCLUDE Typical value: /usr/local/include (no default).

Specifies the path where the Parallel-NetCDF header files are located.

ESMF_PNETCDF_LIBPATH Typical value: `/usr/local/lib` (no default).

Specifies the path where the Parallel-NetCDF library file is located.

ESMF_PNETCDF_LIBS Typical value: `"-lpnetcdf"`.

Specifies the linker directives needed to link the Parallel-NetCDF library to the application.

9.4.4 PIO

ESMF provides the ability to read and write data in NetCDF format through ParallelIO (PIO), a third-party I/O software library that is integrated into the ESMF library. The following environment variable enables PIO functionality inside of ESMF.

The PIO code depends on MPI I/O support by the underlying MPI implementation for parallel I/O. Almost all current MPI implementations support MPI I/O to the required degree. For NetCDF format support the integrated PIO code depends on `ESMF_NETCDF` (see 9.4.2) being enabled and optionally `ESMF_PNETCDF` (see 9.4.3) being enabled.

ESMF_PIO Possible value: *not set* (default), `"internal"`, `"external"`, `"OFF"`.

not set **(default)** PIO-dependent features will be enabled on supported platforms, as determined by the ESMF build configuration.

`"internal"` PIO-dependent features will be enabled and will use the PIO library that is included and built with ESMF. Internal builds of PIO require CMake version 2.8.12 or newer be available in the path.

`"external"` PIO-dependent features will be enabled and will use an external PIO library. The additional parameters `ESMF_PIO_INCLUDE` (path to PIO include files) and `ESMF_PIO_LIBPATH` (path to PIO library files) should also be set when using this option. The minimum version of PIO for this option is 2.5.10.

`"OFF"` Disables PIO-dependent code.

ESMF_PIO_INCLUDE (no default)

Specifies the path where the PIO header files are located.

ESMF_PIO_LIBPATH (no default)

Specifies the path where the PIO library is located.

9.4.5 Accelerator Software Stacks

ESMF provides the ability to query various third party accelerator software stacks and gather information about the accelerator devices available in a system. The users can query the number of accelerator devices accessible from a PET using the OpenCL, OpenACC, Intel MIC and OpenMP software stacks.

The following environment variables enable, and specify the name and location of the desired accelerator software stacks and associated header files:

ESMF_ACC_SOFTWARE_STACK Possible value: *not set* (default), `"opencl"`, `"openacc"`, `"intelmic"`, `"openmp4"`.

not set **(default)** All accelerator software stack related features will be disabled. The `ESMF_ACC_SOFTWARE_STACK_INCLUDE`, `ESMF_ACC_SOFTWARE_STACK_LIBPATH`, and `ESMF_ACC_SOFTWARE_STACK_LIBS` environment variables will be ignored.

"opencl" The ESMF library will use the OpenCL framework to query information about accelerator devices in the system. The ESMF_ACC_SOFTWARE_STACK_INCLUDE, ESMF_ACC_SOFTWARE_STACK_LIBPATH and ESMF_ACC_SOFTWARE_STACK_LIBS environment variables will be used to build and link the library.

"openacc" The ESMF library will use the interfaces defined in the OpenACC standard to query information about accelerator devices in the system. The ESMF_ACC_SOFTWARE_STACK_INCLUDE, ESMF_ACC_SOFTWARE_STACK_LIBPATH and ESMF_ACC_SOFTWARE_STACK_LIBS environment variables are not typically defined since the standard is supported inherently by a OpenACC standard compliant compiler.

"intelmic" The ESMF library will use the interfaces defined by the Intel MIC software stack to query information about accelerator devices in the system. The ESMF_ACC_SOFTWARE_STACK_INCLUDE, ESMF_ACC_SOFTWARE_STACK_LIBPATH and ESMF_ACC_SOFTWARE_STACK_LIBS environment variables are not typically defined since the standard is supported inherently by the Intel compiler.

"openmp4" The ESMF library will use the interfaces defined in the OpenMP v4.0 standard to query information about accelerator devices in the system. The ESMF_ACC_SOFTWARE_STACK_INCLUDE, ESMF_ACC_SOFTWARE_STACK_LIBPATH and ESMF_ACC_SOFTWARE_STACK_LIBS environment variables are not typically defined since the standard is supported inherently by a standard compliant compiler.

ESMF_ACC_SOFTWARE_STACK_INCLUDE (no default)

Specifies the path where the header files for the accelerator software stack is located. If not set, this environment variable is ignored.

ESMF_ACC_SOFTWARE_STACK_LIBPATH (no default)

Specifies the path where the libraries for the accelerator software stack is located. If not set, this environment variable is ignored.

ESMF_ACC_SOFTWARE_STACK_LIBS (no default)

Specifies the linker directives required to link the library with the accelerator software stack. If not set, this environment variable is ignored.

9.4.6 XERCES

ESMF provides the ability to read Attribute data in XML file format via the XERCES C++ library. (Writing Attribute XML files is performed with the standard C++ output file stream facility, rather than with Xerces). The following environment variables enable, and specify the name and location of the desired XERCES C++ library and associated header files:

ESMF_XERCES Possible value: *not set* (default), "standard", <*userstring*>.

not set **(default)** XERCES-dependent features will be disabled. The ESMF_XERCES_INCLUDE, ESMF_XERCES_LIBPATH, and ESMF_XERCES_LIBS environment variables will be ignored.

"standard" ESMF_XERCES_LIBS will be set to "-lxerces-c". The ESMF_XERCES_INCLUDE and ESMF_XERCES_LIBPATH environment variables will also be used, if defined.

<*userstring*> If set, ESMF_XERCES_INCLUDE, ESMF_XERCES_LIBPATH, and ESMF_XERCES_LIBS environment variables will be used, if defined.

ESMF_XERCES_INCLUDE Typical value: /usr/local/include (no default).

Specifies the path where the XERCES C++ header files are located.

ESMF_XERCES_LIBPATH Typical value: `/usr/local/lib` (no default).

Specifies the path where the XERCES C++ library file is located.

ESMF_XERCES_LIBS Typical value: `"-lxerces-c"`.

Specifies the linker directives needed to link the XERCES C++ library to the application.

The default value depends on the setting of `ESMF_XERCES`. For the typical case where `ESMF_XERCES` is set to `"standard"`, `ESMF_XERCES_LIBS` is set to `"-lxerces-c"`.

9.4.7 yaml-cpp

Support for YAML Ain't Markup Language (YAML™) in ESMF is provided through the open-source yaml-cpp library, a YAML parser and emitter written in C++ that implements the YAML Version 1.2 specification.

ESMF provides the option to build yaml-cpp from sources included within the ESMF source tree or to link against an external installation of the library. Note that the internally bundled sources are based on yaml-cpp v0.7.0 and include a patch that adds support for the YAML merge key ("«") syntax. This extension may not be present when using an externally provided installation of yaml-cpp.

The following environment variables control how ESMF accesses the yaml-cpp library:

ESMF_YAMLCPP Possible values: `"internal"` (default), `"standard"`, <*userstring*>.

`"internal"` **(default)** The YAML-dependent code inside of ESMF will be enabled. The yaml-cpp sources included with ESMF will be used to provide YAML support. The `ESMF_YAMLCPP_INCLUDE`, `ESMF_YAMLCPP_LIBPATH`, and `ESMF_YAMLCPP_LIBS` environment variables will be ignored.

`"standard"` The YAML-dependent code inside of ESMF will be enabled. `ESMF_YAMLCPP_LIBS` will be set to `"-lyaml-cpp"` if not set. The `ESMF_YAMLCPP_INCLUDE` and `ESMF_YAMLCPP_LIBPATH` environment variables will also be used, if defined.

<*userstring*> The YAML-dependent code inside of ESMF will be enabled. If set, `ESMF_YAMLCPP_INCLUDE`, `ESMF_YAMLCPP_LIBPATH`, and `ESMF_YAMLCPP_LIBS` environment variables will be used.

ESMF_YAMLCPP_INCLUDE Typical value: `/usr/local/include` (no default).

Specifies the path where the yaml-cpp C++ header files are located.

ESMF_YAMLCPP_LIBPATH Typical value: `/usr/local/lib` (no default).

Specifies the path where the yaml-cpp C++ library file is located.

ESMF_YAMLCPP_LIBS Typical value: `"-lyaml-cpp"`.

Specifies the linker directives needed to link the yaml-cpp C++ library to the application.

The default value depends on the setting of `ESMF_YAMLCPP`. For the typical case where `ESMF_YAMLCPP` is set to `"standard"`, `ESMF_YAMLCPP_LIBS` is set to `"-lyaml-cpp"`.

9.4.8 MOAB

The Mesh Oriented datABase (MOAB) can be used to build an ESMF unstructured Mesh as an alternative to the "native" ESMF Mesh implementation. The decision to use either MOAB or the native ESMF Mesh implementation is made at run time. This aspect is described in the Reference Manual, section `ESMF_MeshSetMOAB()` .The default

is to use the native ESMF Mesh. ESMF will build an internal version of MOAB by default, but an external MOAB installation can be used if desired. The build parameters covered in this section are used to determine which version of MOAB is available to ESMF.

ESMF_MOAB Possible values: `"internal"` (default), `"OFF"`, `"external"`.

`"internal"` **(default)** The MOAB dependent code inside of ESMF will be enabled. The MOAB sources included with ESMF will be used to provide MOAB support. The `ESMF_MOAB_INCLUDE`, `ESMF_MOAB_LIBPATH`, and `ESMF_MOAB_LIBS` environment variables will be ignored.

`"OFF"` Disables MOAB dependent code.

`"external"` The MOAB dependent code inside of ESMF will be enabled. The `ESMF_MOAB_INCLUDE`, `ESMF_MOAB_LIBPATH` and `ESMF_MOAB_LIBS` environment variables must also be specified.

ESMF_MOAB_INCLUDE Typical value: `/usr/local/include` (no default).

Specifies the path where the MOAB C++ header files are located.

ESMF_MOAB_LIBPATH Typical value: `/usr/local/lib` (no default).

Specifies the path where the MOAB C++ library file is located.

ESMF_MOAB_LIBS Typical value: `"-lMOAB"` (no default).

Specifies the linker directives needed to link the MOAB C++ library to the application.

9.4.9 NUMA

The LibNUMA API for Non Uniform Memory Access (NUMA) can be used to discover the NUMA architecture at run-time.

ESMF_NUMA Possible values: `"ON"`, `"standard"`, `"OFF"` (default).

`"ON"`/`"standard"` The NUMA dependent code inside of ESMF will be enabled. The `ESMF_NUMA_LIBS` environment variable will be set to `"-lnuma"`.

`"OFF"` **(default)** Disables NUMA dependent code.

ESMF_NUMA_INCLUDE Typically not needed. (no default).

Specifies the path where the NUMA header files are located.

ESMF_NUMA_LIBPATH Typically not needed. (no default).

Specifies the path where the NUMA library file is located.

ESMF_NUMA_LIBS Typical value: `"-lnuma"`.

Specifies the linker directives needed to link the NUMA library to the application.

9.4.10 NVML

The NVIDIA Management Library (NVML) can be used to discover NVIDIA GPUs that are accessible at run-time.

ESMF_NVML Possible values: `"ON"`, `"standard"`, `"OFF"` (default).

"ON"/"standard" The NVML dependent code inside of ESMF will be enabled. The ESMF_NVML_LIBS environment variable will be set to "-lnvidia-ml".

"OFF" **(default)** Disables NVML dependent code.

ESMF_NVML_INCLUDE Typically not needed. (no default).

Specifies the path where the NVML header files are located.

ESMF_NVML_LIBPATH Typically not needed. (no default).

Specifies the path where the NVML library file is located.

ESMF_NVML_LIBS Typical value: "-lnvidia-ml".

Specifies the linker directives needed to link the NVML library to the application.

9.5 ESMF Environment Variables

The following is a full alphabetical list of all environment variables which are used by the ESMF build system. The ESMF_DIR must be set in all circumstances, while most other environment variables have defaults. However, it is recommended to explicitly set the compiler and MPI flavor using ESMF_COMPILER and ESMF_COMM, respectively, to ensure the expected behavior.

ESMF_ABI Possible value: 32, 64, x86_64_32, x86_64_small, x86_64_medium

If a system supports 32-bit and 64-bit (pointer wordsize) application binary interfaces (ABIs), this variable can be set to select which ABI to use. Valid values are 32 or 64. By default the most common ABI is chosen. On x86_64 architectures three additional, more specific ABI settings are available, x86_64_32, x86_64_small and x86_64_medium.

ESMF_ARRAY_LITE Possible value: TRUE, FALSE (default)

Not normally set by user. ESMF auto-generates subroutine interfaces for a wide variety of data arrays of different ranks, shapes, and types. Setting this variable to TRUE instructs ESMF to *not* generating interfaces for 5D, 6D, and 7D arrays. This shrinks the amount of autogenerated code as well as the number of overloaded interfaces.

ESMF_BOPT Possible value: g, O (default)

This environment variable controls the build option. To make a debuggable version of the library set ESMF_BOPT to g before building. The default is O (capital oh) which builds an optimized version of the library. If ESMF_BOPT is O, ESMF_OPTLEVEL can also be set to a numeric value between 0 and 4 to select a specific optimization level.

ESMF_C Possible value: *executable*

This variable can be used to override the default C compiler and linker front-end executables. The executable may be specified with absolute path overriding the location determined by default from the user's PATH variable.

ESMF_COMM Possible value: *system-dependent*

On systems with a vendor-supplied MPI communications library, the vendor library is chosen by default for communications. On these systems ESMF_COMM is set to mpi, signaling to the ESMF build system to use the vendor MPI implementation. For other systems (e.g. Linux or Darwin) where a multitude of MPI implementations are available, ESMF_COMM must be set to indicate which implementation is used to build the ESMF library. Set ESMF_COMM according to your situation to: mpt, mpich (version 3 and up), mpich1, mpich2, mvapich (all versions), lam, openmpi, or intelmpi. ESMF_COMM

may also be set to `user` indicating that the user will set all the required flags using advanced ESMF environment variables. Some individual MPI builds may create additional libraries that need to be linked in, such as the legacy C++ bindings. These may be specified via the `ESMF_CXXLINKLIBS` and `ESMF_F90LINKLIBS` environment variables.

Alternatively, ESMF comes with a single-processor MPI-bypass library which is the default for Linux and Darwin systems. To force the use of this bypass library set `ESMF_COMM` equal to `mpiuni`.

ESMF_COMPILER Possible value: *system-dependent*

The ESMF library build requires a working Fortran90 and C++ compiler. On platforms that don't come with a single vendor supplied compiler suite (e.g. Linux or Darwin) it is recommended to explicitly set `ESMF_COMPILER` to the desired compiler flavor. Notice that setting the `ESMF_COMPILER` variable does *not* affect how the compiler executables are located on the system. `ESMF_COMPILER` (together with `ESMF_COMM`) affect the name that is expected for the compiler executables. Furthermore, the `ESMF_COMPILER` setting is used to select compiler and linker flags consistent with the compilers indicated.

By default Fortran and C++ compiler executables are expected to be located in a location contained in the user's `PATH` environment variable. This means that if you cannot locate the correct compiler executable via the `which` command on the shell prompt the ESMF build system won't find it either!

There are advanced ESMF environment variables that can be used to select specific compiler executables by specifying the full path. This can be used to pick specific compiler executables without having to modify the `PATH` environment variable.

Use 'make info' to see which compiler executables the ESMF build system will be using according to your environment variable settings.

To see possible values for `ESMF_COMPILER`, cd to `$ESMF_DIR/build_config` and list the directories there. The first part of each directory name corresponds to the output of 'uname -s' for this platform. The second part contains possible values for `ESMF_COMPILER`. In some cases multiple combinations of Fortran and C++ compilers are possible, e.g. there is `intel` and `intelgcc` available for Linux. Setting `ESMF_COMPILER` to `intel` indicates that both Intel Fortran and C++ compilers are used, whereas `intelgcc` indicates that the Intel Fortran compiler is used in combination with GCC's C++ compiler.

If you do not find a configuration that matches your situation you will need to port ESMF.

ESMF_CXX Possible value: *executable*

This variable can be used to override the default C++ compiler and linker front-end executables. The executable may be specified with absolute path overriding the location determined by default from the user's PATH variable.

ESMF_CXXCOMPILEOPTS Possible value: *list of flags*

Prepend compiler flags to the list of flags the ESMF build system determines.

ESMF_CXXCOMPILEPATHS Possible value: *list of paths, each prepended with -I*

Prepend compiler search paths to the list of search paths the ESMF build system determines.

ESMF_CXXCOMPILER Possible value: *executable*

This variable can be used to override the default C++ compiler front-end executables. The executable may be specified with absolute path overriding the location determined by default from the user's PATH variable.

ESMF_CXXLINKER Possible value: *executable*

This variable can be used to override the default C++ linker front-end executables. The executable may be specified with absolute path overriding the location determined by default from the user's PATH variable.

ESMF_CXXLINKLIBS Possible value: *list of libraries, each prepended with -l*

Prepend libraries to the list of libraries the ESMF build system determines.

ESMF_CXXLINKOPTS Possible value: *list of flags*

Prepend linker flags to the list of flags the ESMF build system determines.

ESMF_CXXLINKPATHS Possible value: *list of paths, each prepended with -L*

Prepend linker search paths to the list of search paths the ESMF build system determines.

ESMF_CXXLINKRPATHS Possible value: *list of paths, each prepended with the correct rpath option*

Prepend linker rpaths to the list of rpaths the ESMF build system determines.

ESMF_CXXOPTFLAG Possible value: *flag*

This variable can be used to override the default C++ optimization flag.

ESMF_CXXSTD Possible value: *integer* or *default* or *sysdefault*

Used to set the C++ language standard. If unset or *default*, the ESMF default C++ language standard is used: C++11. If set to an integer, the integer is used to indicate the respective C++ language standard to the compiler. ESMF does not check whether the integer corresponds to an existing language standard. Setting *sysdefault* results in usage of the compiler specific default C++ language standard. This can lead to build issue if the compiler default is below the level required by ESMF.

ESMF_DEFER_LIB_BUILD Possible value: *ON* (default), *OFF*

This variable can be used to override the deferring of the build of the ESMF library. By default, the library is built after all of the source files have been compiled. This speeds up the build process. It also allows parallel compilation of source code when the -j flag is used with make. Setting this environment variable to `OFF` forces the library to be updated after each individual compilation, thus disabling the ability to use parallel compilation.

ESMF_DIR Possible value: *absolute path*

The environment variable `ESMF_DIR` must be set to the full pathname of the top level ESMF directory before building the framework. This is the only environment variable which is required to be set on all platforms under all conditions.

ESMF_F90 Possible value: *executable*

This variable can be used to override the default Fortran90 compiler and linker front-end executables. The executable may be specified with absolute path overriding the location determined by default from the user's PATH variable.

ESMF_F90COMPILEOPTS Possible value: *list of flags*

Prepend compiler flags to the list of flags the ESMF build system determines.

ESMF_F90COMPILEPATHS Possible value: *list of paths, each prepended with -I*

Prepend compiler search paths to the list of search paths the ESMF build system determines.

ESMF_F90COMPILER Possible value: *executable*

This variable can be used to override the default Fortran90 compiler front-end executables. The executable may be specified with absolute path overriding the location determined by default from the user's PATH variable.

ESMF_F90IMOD Possible value: *flag*

This variable can be used to override the default flag (-I) used to specify a Fortran module directory.

ESMF_F90LINKER Possible value: *executable*

This variable can be used to override the default Fortran90 linker front-end executables. The executable may be specified with absolute path overriding the location determined by default from the user's PATH variable.

ESMF_F90LINKLIBS Possible value: *list of libraries, each prepended with -l*

Prepend libraries to the list of libraries the ESMF build system determines.

ESMF_F90LINKOPTS Possible value: *list of flags*

Prepend linker flags to the list of flags the ESMF build system determines.

ESMF_F90LINKPATHS Possible value: *list of paths, each prepended with -L*

Prepend linker search paths to the list of search paths the ESMF build system determines.

ESMF_F90LINKRPATHS Possible value: *list of paths, each prepended with the correct rpath option*

Prepend linker rpaths to the list of rpaths the ESMF build system determines.

ESMF_F90OPTFLAG Possible value: *flag*

This variable can be used to override the default Fortran90 optimization flag.

ESMF_INSTALL_BINDIR Possible value: *relative or absolute path*

Location into which to install the ESMF apps during installation. This location can be specified as absolute path (starting with "/") or relative to `ESMF_INSTALL_PREFIX`.

ESMF_INSTALL_DOCDIR Possible value: *relative or absolute path*

Location into which to install the documentation during installation. This location can be specified as absolute path (starting with "/") or relative to `ESMF_INSTALL_PREFIX`.

ESMF_INSTALL_HEADERDIR Possible value: *relative or absolute path*

Location into which to install the header files during installation. This location can be specified as absolute path (starting with "/") or relative to `ESMF_INSTALL_PREFIX`.

ESMF_INSTALL_LIBDIR Possible value: *relative or absolute path*

Location into which to install the library files during installation. This location can be specified as absolute path (starting with "/") or relative to `ESMF_INSTALL_PREFIX`.

ESMF_INSTALL_MODDIR Possible value: *relative or absolute path*

Location into which to install the F90 module files during installation. This location can be specified as absolute path (starting with "/") or relative to `ESMF_INSTALL_PREFIX`.

ESMF_INSTALL_CMAKEDIR Possible value: *relative or absolute path*

Location into which to install the CMake module files during installation. This location can be specified as absolute path (starting with "/") or relative to `ESMF_INSTALL_PREFIX`.

ESMF_INSTALL_PREFIX Possible value: *relative or absolute path*

This variable specifies the prefix of the installation path used during the installation process accessible thought the install target. Libraries, F90 module files, header files and documentation all are installed relative to `ESMF_INSTALL_PREFIX` by default. The `ESMF_INSTALL_PREFIX` may be provided as absolute path (starting with "/") or relative to `ESMF_DIR`.

ESMF_LAPACK See 9.4.1

ESMF_LAPACK_LIBPATH See 9.4.1

ESMF_LAPACK_LIBS See 9.4.1

ESMF_MACHINE Possible value: output of `uname -m` where available.

Not normally set by user. This variable indicates architectural details about the machine on which the ESMF library is being built. The value of this variable will affect which ABI settings are available and what they mean. `ESMF_MACHINE` is set automatically.

ESMF_MPIBATCHOPTIONS Possible value: *system-dependent*

Variable used to pass system-specific queue options to the batch system. Typically the queue, project and limits are set. See section 11.1.1 for a discussion of this option.

ESMF_MPILAUNCHOPTIONS Possible value: *system-dependent*

Variable used to pass system-specific options to the MPI launch facility. See section 11.1.1 for a discussion of this option.

ESMF_MPIMPMDRUN Possible value: *executable*

This variable can be used to override the default utility used to launch parallel execution of ESMF test applications in MPMD mode. The executable in `ESMF_MPIMPMDRUN` may be specified with path.

ESMF_MPIRUN Possible value: *executable*

This variable can be used to override the default utility used to launch parallel ESMF test or example applications. The executable in `ESMF_MPIRUN` may be specified with path. See section 11.1.1 for a discussion of this option.

ESMF_MPISCRIPTOPTIONS Possible value: *system-dependent*

Variable used to pass system-specific options to the first level MPI script accessed by ESMF. See section 11.1.1 for a discussion of this option.

ESMF_NETCDF See 9.4.2

ESMF_NETCDF_INCLUDE See 9.4.2

ESMF_NETCDF_LIBPATH See 9.4.2

ESMF_NETCDF_LIBS See 9.4.2

ESMF_NO_INTEGER_1_BYTE Possible value: `TRUE` (default), `FALSE`

Not normally set by user. Setting this variable to `FALSE` instructs ESMF to generating data array interfaces for data types of 1-byte integers.

ESMF_NO_INTEGER_2_BYTE Possible value: `TRUE` (default), `FALSE`

Not normally set by user. Setting this variable to `FALSE` instructs ESMF to generating data array interfaces for data types of 2-byte integers.

ESMF_OPENACC Possible value: `ON`, `OFF` (default is system dependent)

Compiles and links the ESMF library with OpenACC compiler flags.

ESMF_OPENMP Possible value: `OMP4`, `ON`, `OFF` (default is system dependent)

Compiles and links the ESMF library with OpenMP compiler flags. Both `OMP4` and `ON` enable the ESMF OpenMP features. Only with `OMP4` will those features assume OpenMP 4.0 and higher.

ESMF_OPTLEVEL Possible value: *numerical value*

See `ESMF_BOPT` for details.

ESMF_OS Possible value: output of `uname -s` except when cross-compiling or for `UNICOS/mp` where `ESMF_OS` is `Unicos`.

Not normally set by user unless cross-compiling. This variable indicates the target system for which the ESMF library is being built. Under normal circumstances, i.e. ESMF is being build on the target system, `ESMF_OS` is set automatically. However, when cross-compiling for a different target system `ESMF_OS` must be set to the respective target OS. For example, when compiling for the Cray X1 on an interactive X1 node `ESMF_OS` will be set automatically. However, when ESMF is being cross-compiled for the X1 on a Linux host the user must set `ESMF_OS` to `Unicos` manually in order to indicate the intended target platform.

ESMF_PNETCDF See 9.4.3

ESMF_PNETCDF_INCLUDE See 9.4.3

ESMF_PNETCDF_LIBPATH See 9.4.3

ESMF_PNETCDF_LIBS See 9.4.3

ESMF_PTHREADS Possible value: `ON` (default on most platforms), `OFF`

This compile-time option controls ESMF's dependency on a functioning Pthreads library. The default option is set to `ON` with the exception of IRIX64 and platforms that don't provide Pthreads. On IRIX64 the use of Pthreads in ESMF is disabled by default because the Pthreads library conflicts with the use of OpenMP on this platform.

The user can override the default setting of `ESMF_PTHREADS` on all platforms that provide Pthread support. Setting the `ESMF_PTHREADS` environment variable to `OFF` will disable ESMF's Pthreads feature set. On platforms that don't support Pthreads, e.g. IBM BlueGene/L or Cray XT3, the default `OFF` setting cannot be overridden!

ESMF_SITE Possible value: *site string*, `default`

Build configure file site name or the value default. If not set, then the value of default is assumed. When including platform-specific files, this value is used as the third part of the directory name (parts 1 and 2 are the ESMF_OS value and ESMF_COMPILER value, respectively.)

ESMF_TESTESMFMKFILE Possible value: `ON`, `OFF` (default)

Variable specifying whether the `ESMFMKFILE` variable is evaluated to determine which ESMF installation is being tested against. If set to the value `ON`, all tests and examples are build against the ESMF installation referenced by the `ESMFMKFILE` variable. For `OFF`, the `ESMFMKFILE` variable is ignored and the tests and examples are build against the ESMF under `ESMF_DIR`. This is the default.

ESMF_TESTEXHAUSTIVE Possible value: `ON`, `OFF` (default)

Variable specifying how to compile the unit tests. If set to the value `ON`, then all unit tests will be compiled and will be executed when the test is run. If unset or set to any other value, only a subset of the unit tests will be included to verify basic functions. Note that this is a compile-time selection, not a run-time option.

ESMF_TESTFORCEOPENACC Possible value: `ON`, `OFF` (default)

The `ON` setting enforces usage of OpenACC compiler flags when building ESMF test applications. This allows testing of user-level OpenACC usage even with `ESMF_OPENACC` set to `OFF`.

ESMF_TESTFORCEOPENMP Possible value: `ON`, `OFF` (default)

The `ON` setting enforces usage of OpenMP compiler flags when building ESMF test applications. This allows testing of user-level OpenMP usage even with `ESMF_OPENMP` set to `OFF`.

ESMF_TESTHARNESS_ARRAY Possible value: *test harness make target* (default not set)

Variable specifying the test harness makefile target for the array class. If this variable is not specified, a default test scenario will be run for the array class. See the ESMF Software Developer's Guide for instructions for selecting other test harness scenarios.

ESMF_TESTHARNESS_FIELD Possible value: *test harness make target* (default not set)

Variable specifying the test harness makefile target for the field class. If this variable is not specified, a default test scenario will be run for the field class. See the ESMF Software Developer's Guide for instructions for selecting other test harness scenarios.

ESMF_TESTMPMD Possible value: `ON`, `OFF` (default)

Variable specifying whether to run MPMD-style tests, i.e. test applications that start up as multiple separate executables.

ESMF_TESTSHAREDOBJ Possible value: `ON`, `OFF` (default)

Variable specifying whether to run shared object tests. This requires that the compute environment supports shared objects, and that the ESMF library is available in form of a shared library.

ESMF_TESTWITHTHREADS Possible value: `ON`, `OFF` (default)

If this environment variable is set to `ON` *before* the ESMF system tests are build they will activate ESMF threading in their code. Specifically each component will be executed using ESMF single threading instead of the default non-threaded mode. The difference between non-threaded and ESMF single threaded execution should be completely transparent. Notice that the setting of `ESMF_TESTWITHTHREADS` does *not* alter ESMF's dependency on Pthreads but tests ESMF threading features during the system tests. An ESMF library that was compiled with disabled Pthread features (via the `ESMF_PTHREADS` variable) will produce ESMF error messages during system test execution if the system tests were compiled with `ESMF_TESTWITHTHREADS` set to `ON`.

ESMF_TRACE_LIB_BUILD Possible value: *ON* (default), *OFF*

This variables determines whether extra libraries are built that are used to add additional symbols to the ESMF tracing and profiling capability, such as MPI communication functions. If set to `ON` the libraries are built and placed into the `ESMF_INSTALL_LIBDIR` alongside the ESMF library itself.

ESMF_XERCES See 9.4.6

ESMF_XERCES_INCLUDE See 9.4.6

ESMF_XERCES_LIBPATH See 9.4.6

ESMF_XERCES_LIBS See 9.4.6

ESMF_YAMLCPP See 9.4.7

ESMF_YAMLCPP_INCLUDE See 9.4.7

ESMF_YAMLCPP_LIBPATH See 9.4.7

ESMF_YAMLCPP_LIBS See 9.4.7

Environment variables must be set in the user's shell or when calling make. It is *not* necessary to edit ESMF makefiles or other build system files to set these variables. Here is an example of setting an environment variable in the csh/tcsh shell:

```
setenv ESMF_ABI 32
```

In bash/ksh shell environment variables are set this way:

```
export ESMF_ABI=32
```

Environment variables can also be set from the make command line:

```
make ESMF_ABI=32
```

9.6 Supported Platforms

The platforms that are tested and supported depend on the release of ESMF. To see the specific list of supported platforms, click the *Supported Platforms* link under one of the releases on the ESMF releases page.

All *possible* combinations of ESMF_OS, ESMF_COMPILER, ESMF_COMM, and ESMF_ABI build environment variables are listed in the following table. Where multiple options exist, the default value is indicated in **bold**. An entry of default in the COMPILER column indicates the vendor compiler. An entry of mpi in the COMM column indicates the vendor MPI implementation.

ESMF_OS	ESMF_COMPILER	ESMF_COMM	ESMF_ABI
AIX	**default**	mpiuni,**mpi**,user	32, **64**
Cygwin	g95	**mpiuni**,mpich,mpich1,mpich2,lam,openmpi,user	32, 64
Cygwin	**gfortran**	**mpiuni**,mpich,mpich1,mpich2,lam,msmpi,openmpi,user	32, 64
Darwin	absoft	**mpiuni**,mpich,mpich1,mpich2,mvapich,lam,openmpi,user	32, 64
Darwin	g95	**mpiuni**,mpich,mpich1,mpich2,mvapich,lam,openmpi,user	32, 64
Darwin	gfortran	**mpiuni**,mpich,mpich1,mpich2,mvapich,lam,openmpi,user	32, 64
Darwin	**gfortranclang**	**mpiuni**,mpich,mpich1,mpich2,mvapich,lam,openmpi,user	32, 64
Darwin	intel	**mpiuni**,mpich,mpich1,mpich2,mvapich,intelmpi,lam, openmpi,user	32, 64
Darwin	intelclang	**mpiuni**,mpich,mpich1,mpich2,intelmpi,lam,openmpi,user	32, 64
Darwin	intelgcc	**mpiuni**,mpich,mpich1,mpich2,intelmpi,lam,openmpi,user	32, 64
Darwin	llvm	**mpiuni**,mpich,mpich1,mpich2,mvapich,openmpi,user	32, 64
Darwin	nag	**mpiuni**,mpich,mpich1,mpich2,mvapich,lam,openmpi,user	32, 64
Darwin	nagclang	**mpiuni**,mpich,mpich1,mpich2,mvapich,lam,openmpi,user	32, 64
Darwin	pgi	**mpiuni**,mpich,mpich1,mpich2,mvapich,lam,openmpi,user	32, 64
Darwin	xlf	mpiuni,**mpi**,mpich,mpich1,mpich2,lam,openmpi,user	32
Darwin	xlfgcc	mpiuni,**mpi**,mpich,mpich1,mpich2,lam,openmpi,user	32
IRIX64	**default**	mpiuni,**mpi**,user	32, **64**
Linux	absoft	**mpiuni**,mpich,mpich1,mpich2,mvapich,lam,openmpi,user	32, 64
Linux	absoftintel	**mpiuni**,mpich,mpich1,mpich2,lam,openmpi,user	32, 64
Linux	aocc	**mpiuni**,mpi,mpt,mpich,mpich1,mpich2,mvapich, intelmpi,lam,openmpi,user	32, 64, ia64_64, x86_64_32, x86_64_small, x86_64_medium
Linux	arm	**mpiuni**,mpi,mpt,mpich,mpich1,mpich2,mvapich, intelmpi,lam,openmpi,user	32, 64, ia64_64, x86_64_32,

			x86_64_small, x86_64_medium
Linux	fujitsu	**mpiuni**,mpi,user	64
Linux	g95	**mpiuni**,mpich,mpich1,mpich2,mvapich,lam,openmpi,user	32, 64, ia64_64, x86_64_32, x86_64_small, x86_64_medium
Linux	**gfortran**	**mpiuni**,mpi,mpt,mpich,mpich1,mpich2,mvapich, intelmpi,lam,openmpi,user	32, 64, ia64_64, x86_64_32, x86_64_small, x86_64_medium
Linux	gfortranclang	**mpiuni**,mpi,mpt,mpich,mpich1,mpich2,mvapich, lam,openmpi,user	32, 64, ia64_64, x86_64_32, x86_64_small, x86_64_medium
Linux	intel	**mpiuni**,mpi,mpt,mpich,mpich1,mpich2,mvapich, intelmpi,scalimpi,lam,openmpi,user	32, 64, ia64_64, x86_64_32, x86_64_small, x86_64_medium, mic
Linux	intelgcc	**mpiuni**,mpi,mpt,mpich,mpich1,mpich2,mvapich, intelmpi,lam,openmpi,user	32, 64, ia64_64, x86_64_32, x86_64_small, x86_64_medium
Linux	lahey	**mpiuni**,mpich,mpich1,mpich2,mvapich,lam, openmpi,user	32, 64
Linux	llvm	**mpiuni**,mpi,mpt,mpich,mpich1,mpich2,mvapich, intelmpi,lam,openmpi,user	32, 64, ia64_64, x86_64_32, x86_64_small, x86_64_medium
Linux	nag	**mpiuni**,mpich,mpich1,mpich2,mvapich,lam, openmpi,user	32, 64
Linux	nagintel	**mpiuni**,mpich,mpich1,mpich2,lam,openmpi,user	32, 64
Linux	nvhpc	**mpiuni**,mpi,mpt,mpich,mpich1,mpich2,mvapich, intelmpi,openmpi,user	32, 64, x86_64_32, x86_64_small, x86_64_medium
Linux	pathscale	**mpiuni**,mpich,mpich1,mpich2,lam,openmpi,user	32, 64, x86_64_32, x86_64_small, x86_64_medium
Linux	pgi	**mpiuni**,mpi,mpt,mpich,mpich1,mpich2,mvapich, intelmpi,scalimpi,lam,openmpi,user	32, 64, x86_64_32, x86_64_small, x86_64_medium

Linux	pgigcc	**mpiuni**,mpich,mpich1,mpich2,lam,openmpi,user	32
Linux	sxcross	mpiuni,**mpi**,user	32
Linux	xlf	mpiuni,**mpi**,user	32
MinGW	**gfortran**	**mpiuni**,msmpi,user	32, 64
MinGW	intel	**mpiuni**,msmpi,user	32, 64
MinGW	intelcl	**mpiuni**,msmpi,user	32, 64
OSF1	**default**	mpiuni,**mpi**,user	64
SunOS	**default**	mpiuni,**mpi**,user	32, **64**
Unicos	default	mpiuni,**mpi**,user	64
Unicos	aocc	mpiuni,**mpi**,user	64
Unicos	cce	mpiuni,**mpi**,user	64
Unicos	gfortran	mpiuni,**mpi**,user	64
Unicos	intel	mpiuni,**mpi**,user	64
Unicos	nvhpc	mpiuni,**mpi**,user	64
Unicos	pathscale	mpiuni,**mpi**,user	64
Unicos	**pgi**	mpiuni,**mpi**,user	64

Building the library for multiple architectures or options at the same time is supported; building or running the tests or examples is restricted to one platform/architecture at a time. The output from the test cases will be stored in a separate directories so the results will be kept separate for different architectures or options.

9.7 Building the ESMF Library

GNU Make is required to build the ESMF library. On some systems this will be just the command `make`. On others it might be installed as `gmake` or `gnumake`. This document uses `make` consistently to refer to GNU Make.

Use the `--version` option with the locally available make commands to determine which variant corresponds to GNU Make on your system. Use the respective command when interacting with the ESMF build system, and where this documentation uses `make`.

Notice that ESMF does not utilize Autotools (configure or autoconf) or CMake. Instead, the selection of configuration options is done by setting environment variables before building the framework. The relevant environment variables all begin with prefix `ESMF_`, and are discussed in detail under section 9.5.

Build the library with the command:

```
make
```

Makefiles throughout the framework are configured to allow users to compile files only in the directory where `make` is entered. Shared libraries are rebuilt only if necessary. In addition the entire ESMF framework may be built from any directory by entering `make all`, assuming that all the environmental variables are set correctly as described in Section 9.5.

The makefiles are also configured to allow multiple make targets to be compiled in parallel, via the make -j flag. For example, to use eight parallel processes to build the library, use -j8:

```
make -j8 lib
```

The parallel compilation feature depends on `ESMF_DEFER_LIB_BUILD=ON` (the default) so that the library build will be deferred until all files have been compiled.

The -j option should only be used during the creation of the library. The test base and examples will not work correctly with -j set larger than 1.

Users may also run examples or execute unit tests of specific classes by changing directories to the desired class `examples` or `tests` directories and entering `make run_examples` or `make run_unit_tests`, respectively. For non-multiprocessor machines, uni-processor targets are available as `make run_examples_uni` or `make run_unit_tests_uni`.

9.8 Building the ESMF Documentation

The ESMF source documentation consists of an *ESMF User's Guide* and an *ESMF Reference Manual for Fortran*.

If a user does want to build the documentation, they will need to download the ESMF code repository (see section 5.1). Latex, latex2html, perl and csh must also be installed. For example, dependencies may be installed on Ubuntu Linux using:

```
[sudo] apt-get install texlive latex2html perl csh
```

To build documentation:

```
make doc              ! Builds the manuals, including pdf and html.
```

The resulting documentation files will be located in the top level directory $ESMF_DIR/doc

9.9 Installing the ESMF

The ESMF build system offers the standard `install` target to install all necessary files created during the build process into user specified locations. The installation procedure will also install the ESMF documentation if it has been built successfully following the procedure outlined above.

The installation location can be customized using six `ESMF_` environment variables:

- `ESMF_INSTALL_PREFIX` – prefix for the other five variables.
- `ESMF_INSTALL_HEADERDIR` – where to install header files.
- `ESMF_INSTALL_LIBDIR` – where to install library files.
- `ESMF_INSTALL_MODDIR` – where to install Fortran module files.
- `ESMF_INSTALL_BINDIR` – where to install application files.
- `ESMF_INSTALL_DOCDIR` – where to install documentation files.
- `ESMF_INSTALL_CMAKEDIR` – where to install cmake module files.

Section 9.5 describes what each of these environment variables does and how to set them.

Install ESMF with the command:

```
make install
```

Check the ESMF installation with the command:

```
make installcheck
```

Advice to installers. To complete the installation of ESMF, a single ESMF specific environment variable should be set. The variable is named ESMFMKFILE, and it must point to the esmf.mk file that was generated during the installation process. Systems that support multiple ESMF installations via management software (e.g. *modules, softenv, ...*) should set/reset the ESMFMKFILE environment variable as part of the configuration.

Additionally, it is typically convenient to append the user's PATH environment variable to provide access to the ESMF applications that were built during the installation process. The application binaries are located in the directory that was specified as ESMF_INSTALL_BINDIR during the ESMF installation. The location is also stored in variable ESMF_APPSDIR, defined in file esmf.mk. Systems that make ESMF installations available through management software (e.g. *modules, softenv, ...*) should modify the user's PATH environment variable as part of the configuration.

Hint. By default, file esmf.mk is located next to the ESMF library file in directory ESMF_INSTALL_LIBDIR. Consequently, unless esmf.mk has been moved to a different location after the installation, the correct setting for ESMFMKFILE is $(ESMF_INSTALL_LIBDIR)/esmf.mk.

Rationale. The only piece of information that is needed to use an ESMF installation is the exact location of the associated esmf.mk file. This file contains all of the relevant settings and flags that allow a user to build their application against the ESMF installation. Standardizing the mechanism by which the location of esmf.mk is made available to the user by the system will help users in the design of portable application build systems. (See sections 6 and 8 for details about the usage of esmf.mk.) Further, modifying the user's PATH environment variable is optional, since the location of the ESMF application binaries is available through the esmf.mk file. However, setting the user's PATH variable so that the ESMF applications are directly and conveniently accessible from the command line is recommended, especially if management software (e.g. *modules, softenv, ...*) is used on the system.

10 Porting ESMF

This section goes into more detail about the ESMF build system and how to port the ESMF software to new platforms.

10.1 The ESMF Build System

For most users the description of the build system in previous sections should be sufficient. Some users, however, may wish to have a more detailed knowledge of the make system either for configuring different build options or for porting to unsupported platforms.

10.1.1 General structure

The main components of the build system are:

- **Build directories with makefile fragments**

  There are two directories containing makefile fragment files used by the ESMF build system.

  The build directory contains the generic makefile fragment file common.mk that is included by the top level makefile in the source tree. The common.mk contains generic build system settings and build rules used across all platforms. A user should have no reason to edit common.mk.

The build_config directory contains subdirectories with makefile fragments (build_rules.mk) for each supported platform defining compilers, compiler flags and the various other definitions that are necessary to build on each platform. One of the build_rules.mk files will be included by the build/common.mk file depending on the values of the environment variables ESMF_OS, ESMF_COMPILER and ESMF_SITE. See below for more details on environment variables.

- **Environment variables**

  Environment variables with the prefix ESMF_ are used to pass user specified information to the ESMF build system. A full list of ESMF_ environment variables is provided in section 9.5 of this document.

  Most environment variables are optional and the ESMF build system will use default settings if it finds these variable unset. One piece of information that must always be provided by setting the respective environment variable is the root of the ESMF directory. There are three sets of source codes the build system supports. All need environment variables set to point to their top level source code directories.

  ESMF Library
  : To build the ESMF library, ESMF_DIR needs to be set to the top level ESMF library source code directory.

  Implementation Report
  : The build system needs ESMF_IMPL_DIR set to the top level source code directory of the Implementation Report source tree to build the report and to build and run the examples.

  EVA Applications
  : An EVA source code tree does not contain a copy of the ESMF build system. Instead it uses a copy found in an ESMF library source code tree. Building the EVA applications requires that ESMF_EVA_DIR and ESMF_DIR be set. ESMF_EVA_DIR has to be set to the top directory of the EVA source code. ESMF_DIR has to be set to the top directory of an ESMF source code tree.

- **Makefiles**

  Every source tree contains a makefile in its top level directory. This makefile includes the common.mk file from the build directory which in turn includes the platform specific build_rules.mk file from one of the build_config subdirectories. The top level makefile contains makefile settings specific for the source code that it is found in.

  Each directory in the source tree contains a makefile which includes the top level makefile. These local makefiles include definitions that allow the local files and documents to be built.

10.1.2 Build configuration

A single makefile or makefile fragment from the build system never constitutes a complete set of build rules and settings. Starting from the local makefile, successive include commands are used to string together makefiles and makefile fragments to create a complete system of build rules and settings. Configuration of the build system is done by including a configuration makefile fragment. A configuration for a specific machine or compiler is referred to as a site configuration.

The string of files included is fairly short. Makefiles below the top level makefile include the top level makefile. The top level makefile includes build/common.mk and then build/common.mk includes a configuration file from the build_config directory. The configuration files in the build_config directory contain the platform and site specific build settings. The os, compiler and site that a file configures is determined by its name. The configuration makefile fragments follow the naming convention

```
build_config/ESMF_OS.ESMF_COMPILER.ESMF_SITE/build_rules.mk
```

where ESMF_OS, ESMF_COMPILER and ESMF_SITE are environment variables either set by the user or given default values by the build system. ESMF_OS is the target operating system. If the build is performed *on* the target system ESMF_OS will typically have the value returned by the command uname -s. ESMF_COMPILER is the compiler name. ESMF_SITE, if set, is generally the current machine name, the location, or the organization (e.g. mit, cola). If there are no site specific files for a particular platform, then ESMF_COMPILER and ESMF_SITE will be set to default. Examples:

```
! Default configuration for IBM AIX systems
build_config/AIX.default.default/build_rules.mk

! Linux configuration using lahey compilers.
build_config/Linux.lahey.default/build_rules.mk
```

10.1.3 Source code configuration

Some of the ESMF C++ and Fortran source files contain preprocessor directives to configure the source code for specific platforms. The directives are included in the source code and are pre-processed before the source code is compiled. The directives are used to determine among other things, the size of variable types.

The ESMF build system provides preprocessor directives in ESMC_Conf.h and ESMF_Conf.inc files that are included in the source code. These files are located in

```
build_config/ESMF_OS.ESMF_COMPILER.ESMF_SITE/ESMC_Conf.h
build_config/ESMF_OS.ESMF_COMPILER.ESMF_SITE/ESMF_Conf.inc
```

where ESMF_OS, ESMF_COMPILER and ESMF_SITE are environment variables set by the user or given default values be the build system. Based on the settings of these environment variables the build system provides a path to the correct files during source code compilation.

10.2 Porting ESMF to New Platforms

The ESMF build system can be ported to other Unix platforms by adding a new platform specific makefile fragment and two associated configuration files. These files (build_rules.mk, ESMC_Conf.h, ESMF_Conf.inc) must be placed into a new subdirectory of the build_config directory, following the ESMF_OS.ESMF_COMPILER.ESMF_SITE naming convention.

When porting to a new platform it is often helpful to start with a copy of the configuration of an existing ESMF port. You may, for example, want to start with a copy of the build_config/Linux.g95.default directory when working on a new Linux configuration.

10.2.1 Customizing the build_rules.mk fragment

The purpose of the build_rules.mk makefile fragment is to customize the build procedure for a specific platform. The customization is done via makefile variables. The main makefile at the top level of the ESMF directory structure first includes the common.mk makefile fragment. This common makefile fragment defines a large number of variables, setting them either to generally valid default values or to specific values the user has set in their environment using ESMF_ style environment variables.

The platform specific build_rules.mk makefile fragment is included by common.mk *after* the variables have been initialized, but *before* any rules are defined in common.mk using these variables. This gives build_rules.mk a chance to modify these variables as it may be necessary to accommodate platform specific properties.

Fortunately only a very small subset of variables pre-defined in common.mk typically need to be modified or overridden in build_rules.mk with platform specific settings. However, there are some variables that *must* be set in every build_rules.mk file. These are variables that are not pre-set in common.mk.

ESMF_CXXDEFAULT Default C++ compiler to be used on this platform. This variable will be used by common.mk to set the associated ESMF_CXX variables.

ESMF_CXXCOMPILER_VERSION Command that when executed will provide information about the version of the C++ compiler to stdout.

ESMF_F90DEFAULT Default Fortran compiler to be used on this platform. This variable will be used by common.mk to set the associated ESMF_F90 variables.

ESMF_F90COMPILER_VERSION Command that when executed will provide information about the version of the F90 compiler to stdout.

ESMF_MPIRUNDEFAULT Default MPI job launch facility to be used on this platform. This variable will be used by common.mk to set the associated ESMF_MPIRUN variables.

The following is a complete alphabetical list of variables that are pre-set in common.mk before build_rules.mk is included. Some of these variables correspond to ESMF_ environment variables while others have a more complicated dependency on the environment variables set by the user.

ESMF_ABI

ESMF_APPSDIR

ESMF_AR

ESMF_ARCREATEFLAGS

ESMF_ARCREATEFLAGSDEFAULT

ESMF_ARDEFAULT

ESMF_AREXTRACTFLAGS

ESMF_AREXTRACTFLAGSDEFAULT

ESMF_ARRAY_LITE

ESMF_BOPT

ESMF_BUILD

ESMF_BUILD_DOCDIR

ESMF_COMM

ESMF_COMPILER

ESMF_CONFDIR

ESMF_CPP

ESMF_CPPDEFAULT

ESMF_CXXCOMPILECPPFLAGS

ESMF_CXXCOMPILEOPTS

ESMF_CXXCOMPILEPATHS

ESMF_CXXCOMPILEPATHSLOCAL

ESMF_CXXCOMPILER

ESMF_CXXCOMPILERDEFAULT

ESMF_CXXESMFLINKLIBS

ESMF_CXXLINKER

ESMF_CXXLINKERDEFAULT

ESMF_CXXLINKLIBS

ESMF_CXXLINKOPTS

ESMF_CXXLINKPATHS

ESMF_CXXLINKRPATHS

ESMF_CXXOPTFLAG

ESMF_CXXOPTFLAG_G

ESMF_CXXOPTFLAG_O

ESMF_CXXOPTFLAG_X

ESMF_DIR

ESMF_DOCDIR

ESMF_EXDIR

ESMF_F90COMPILECPPFLAGS

ESMF_F90COMPILEFIXCPP

ESMF_F90COMPILEFIXNOCPP

ESMF_F90COMPILEFREECPP

ESMF_F90COMPILEFREENOCPP

ESMF_F90COMPILEOPTS

ESMF_F90COMPILEPATHS

ESMF_F90COMPILEPATHSLOCAL

ESMF_F90COMPILER

ESMF_F90COMPILERDEFAULT

ESMF_F90ESMFLINKLIBS

ESMF_F90IMOD

ESMF_F90LINKER

ESMF_F90LINKERDEFAULT

ESMF_F90LINKLIBS

ESMF_F90LINKOPTS

ESMF_F90LINKPATHS

ESMF_F90LINKRPATHS

ESMF_F90MODDIR

ESMF_F90OPTFLAG

ESMF_F90OPTFLAG_G

ESMF_F90OPTFLAG_O

ESMF_F90OPTFLAG_X

ESMF_GREPV

ESMF_INCDIR

ESMF_INSTALL_BINDIR

ESMF_INSTALL_BINDIR_ABSPATH

ESMF_INSTALL_CMAKEDIR

ESMF_INSTALL_CMAKEDIR_ABSPATH

ESMF_INSTALL_DOCDIR

ESMF_INSTALL_DOCDIR_ABSPATH

ESMF_INSTALL_HEADERDIR

ESMF_INSTALL_HEADERDIR_ABSPATH

ESMF_INSTALL_LIBDIR

ESMF_INSTALL_LIBDIR_ABSPATH

ESMF_INSTALL_MODDIR

ESMF_INSTALL_MODDIR_ABSPATH

ESMF_INSTALL_PREFIX

ESMF_INSTALL_PREFIX_ABSPATH

ESMF_LDIR

ESMF_LIBDIR

ESMF_LOCOBJDIR

ESMF_MACHINE

ESMF_MODDIR

ESMF_MPIBATCHOPTIONS

ESMF_MPILAUNCHOPTIONS

ESMF_MPIMPMDRUN

ESMF_MPIMPMDRUNDEFAULT

ESMF_MPIRUN

ESMF_MPIRUNDEFAULT

ESMF_MPISCRIPTOPTIONS

ESMF_MV

ESMF_NO_INTEGER_1_BYTE

ESMF_NO_INTEGER_2_BYTE

ESMF_OS

ESMF_OPTLEVEL

ESMF_PTHREADS

ESMF_PTHREADSDEFAULT

ESMF_RANLIB

ESMF_RANLIBDEFAULT

ESMF_RM

ESMF_RPATHPREFIX

ESMF_SED

ESMF_SEDDEFAULT

ESMF_SITE

ESMF_SITEDIR

ESMF_SL_LIBLIBS

ESMF_SL_LIBLINKER

ESMF_SL_LIBOPTS

ESMF_SL_LIBS_TO_MAKE

ESMF_SL_SUFFIX

ESMF_STDIR

ESMF_TEMPLATES

ESMF_TESTDIR

ESMF_TESTEXHAUSTIVE

ESMF_TESTMPMD

ESMF_TESTWITHTHREADS

ESMF_UTCDIR

ESMF_UTCSCRIPTS

ESMF_WC

10.2.2 Customizing `ESMC_Conf.h` and `ESMF_Conf.inc`

The `ESMC_Conf.h` file is used to define several settings used during compilation of ESMF library code written in C++.

FTN_X(func) Macro that will correctly expand "func" to match the Fortran symbol convention. Use this macro for function names that contain an underscore.

FTNX(func) Macro that will correctly expand "func" to match the Fortran symbol convention. Use this macro for function names that do *not* contain an underscore.

ESMCI_FortranStrLenArg Typedef to match the data type of the 'hidden' string length argument that Fortran uses when passing CHARACTER strings.

ESMF_PRESENT(arg) Macro for a boolean expression that returns TRUE if "arg" is a "present" argument passed from Fortran into C++.

ESMC_POINTER_SIZE Size of C pointer in bytes.

The `ESMF_Conf.inc` file is used to *optionally* define two important macros:

ESMF_NO_INITIALIZERS If this macro is defined ESMF will assume that initializers inside Fortran derived type definitions are not supported.

ESMF_SEQUENCE_BUG If this macro is defined ESMF will not use the `SEQUENCE` specifier inside Fortran derived types under certain circumstances.

10.3 Shared Object Libraries

On many platforms, a shared object library is created in addition to the standard `.a` archive library. Shared object libraries are libraries that are pre-linked into an executable. They can then be linked to an application at run time. There are many advantages to using shared libraries. These include smaller executable files, and shared memory usage when multiple executables are running - as is often the case of programs using MPI. They also allow easier bug fixing and development because the library can often be upgraded without necessarily re-linking the executables which call into it.

Shared object libraries can be pre-linked to system libraries and using them can simplify dealing with ESMF's dependency on Fortran90 and C++ runtime libraries.

See 9.4 for third party library build requirements.

11 Validating an ESMF Build

The following subsections go into more detail about how to run the tests and examples included with the ESMF software. This is the recommended method of regression testing ESMF, and is routinely used during library code development. Running the regression tests against an existing ESMF installation is also supported, and offers a general way to validate a pre-installed ESMF library.

11.1 Running ESMF Self-Tests

Robustness and portability are primary goals of the ESMF development effort. To ensure that these goals are met, the ESMF includes a comprehensive suite of tests. They allow testing and validation of everything from individual functions to complete system tests. These test suites are used by the ESMF development team as part of their regular development process. ESMF users can run the testing suites to verify that the framework software was built and installed properly, and is running correctly on a particular platform.

11.1.1 Setting up ESMF to run test suite applications

Unless the ESMF library was built in MPI-bypass mode (mpiuni), all applications compiled and linked against ESMF automatically become MPI applications and must be executed as such. The ESMF test suite and example applications are no different in this respect.

Details of how to execute MPI applications vary widely from system to system. ESMF uses an mpirun script mechanism to abstract away most of these differences. All ESMF makefile targets that require the execution of applications do this by launching the application via the executable specified in the `ESMF_MPIRUN` variable. ESMF assumes that an MPI applications can be launched across `N` processes by calling

```
$(ESMF_MPIRUN) -np N application
```

and that the output of the application arrives at the calling shell via `stdout` and `stderr`.

First, on systems that allow direct launching of MPI application via a suitable `mpirun` facility, ESMF can use it directly. This is the ESMF default for all those configurations that come with a suitable `mpirun`. In these cases the `ESMF_MPIRUN` environment variable does not need to be set by the user.

There are systems, however, that allow direct launching of MPI application but provide a launch mechanism that is incompatible with ESMF's assumptions. In these cases a simple mpirun wrapper is required. The ESMF `./scripts` directory contains wrappers for several cases in this class, e.g. for interactive POE access on IBM machines and `aprun`, as well as `yod` on Cray machines. The ESMF configurations will access the appropriate wrapper scripts by default if necessary.

Secondly, there are those systems that utilize batch software to access the parallel execution environment. One option is to execute the ESMF test targets from within a batch session, either interactively or from within a script. In this case the batch software does not add any additional complexity for ESMF. The same issues discussed above, of how to launch an MPI application, apply directly.

However, in some cases it is more convenient to execute the ESMF test target on the front-end machine, and have ESMF access the batch software each time it needs to launch an application. In fact, on IBM systems this is often the only working option, because the integrated POE system will execute each application on the exact same number of processes specified during batch access, regardless of how many ways in parallel a specific application needs to be run.

Two modes of operation need to be considered in the case of the ESMF batch access. First, if interactive batch access is available, it is straightforward to write an `mpirun` script that fulfills the ESMF requirements outlined above. The ESMF `./scripts` directory contains several scripts that access various parallel launching facilities though interactive LSF.

Second, if interactive batch access is not available, a more complex scripting approach is necessary. The basic requirements in this case are that ESMF must be able to launch MPI applications across `N` processes by calling

```
$(ESMF_MPIRUN) -np N application  ,
```

that the output of the application will be available in a file named `application.stdout` after the script finishes, and that the `ESMF_MPIRUN` script blocks execution until `application.stdout` has become accessible.

The ESMF `./scripts` directory contains scripts of this flavor for a wide variety of batch systems. Most of these scripts, when called through ESMF, will generate a customized, temporary batch script for a specific executable "on the fly" and submit this batch script to the queuing software. The script then waits for completion of the submitted job, after which it copies the output, received through a system specific mechanism, into the prescribed file.

Regardless of whether the batch system access is interactive or not, it is often necessary to specify various system specific options when calling the batch submission tool. ESMF utilizes the `ESMF_MPIBATCHOPTIONS` environment variable to pass user supplied values to the batch system.

The environment variable `ESMF_MPISCRIPTOPTIONS` is available to pass user specified options to the actual script specified by `ESMF_MPIRUN`. However, `ESMF_MPISCRIPTOPTIONS` will only be added automatically to the `ESMF_MPIRUN` call if the specified `ESMF_MPIRUN` can be found in the ESMF `./scripts` directory.

Finally, the value of `ESMF_MPILAUNCHOPTIONS` is passed to the MPI launch facility by default, i.e if `ESMF_MPIRUN` was not specified by the user. In case the user specifies `ESMF_MPIRUN` to be anything else but scripts out of the ESMF `./scripts` directory, it is the user's responsibility to add `ESMF_MPISCRIPTOPTIONS` to `ESMF_MPIRUN` and/or to utilize `ESMF_MPILAUNCHOPTIONS` within the specified script.

The possibilities covered by the generic scripts provided in the ESMF `./scripts` directory, combined with the `ESMF_MPISCRIPTOPTIONS`, `ESMF_MPIBATCHOPTIONS`, and `ESMF_MPILAUNCHOPTIONS` environment variables, will satisfy the majority of common situations. There are, however, circumstances for which a customized, user-provided mpirun script is necessary. One such situation arises with the LoadLeveler batch software. LoadLeveler typically requires a list of options specified in the actual batch script. This is most easily handled by a script that produces such a system and user specific script "on the fly". Another situation is where certain modules or software packages need to be made available inside the batch script. Again this is most easily handled by a customized script the user writes and provides to ESMF via the `ESMF_MPIRUN` environment variable. This will override any default settings for the configuration and rely on the user provided script instead.

Users that face the need to write a customized mpirun script for their parallel execution environment are encouraged to start with the closest match from the ESMF `./scripts` directory and customize it to their situation. The best way to see how the existing scripts are used on the supported platforms is to go to the http://www.earthsystemmodeling.org/download/platforms/ web page and follow the link for the platform of interest. Each test report contains the output of `make info`, which lists the settings of the `ESMF_MPIxxx` environment variables.

11.1.2 Running ESMF unit tests

The unit tests provided with the ESMF library evaluate the following:

- correctness of individual functions
- behavior of individual modules or classes
- appropriate error handling

Unit tests can be run in either an exhaustive or a non-exhaustive (sanity check) mode. The exhaustive mode includes the sanity check tests. Typically, sanity checks for each ESMF capability include creating and destroying an object and testing its basic function using a valid argument set. In the exhaustive mode, a wide range of valid and non-valid arguments are evaluated for correct behavior.

The following commands are used to build and run the unit tests provided with the ESMF:

```
make [ESMF_TESTEXHAUSTIVE=<ON,OFF>] unit_tests
make [ESMF_TESTEXHAUSTIVE=<ON,OFF>] unit_tests_uni
```

The `tests_uni` target runs the tests on a single processor. The `tests` target runs the test on multiple processors.

The non-exhaustive set of unit tests should all pass. At this point in development, the exhaustive tests do not all pass. Current problems with unit tests are being tracked and corrected by the ESMF development team.

The results of running the unit tests can be found in the following location:

```
${ESMF_DIR}/test/test${ESMF_BOPT}/${ESMF_OS}.${ESMF_COMPILER}.${ESMF_ABI}. \
${ESMF_SITE}
```

For example, if your esmf source files have been placed in:

```
/usr/local/esmf
```

If your platform is a Linux uni-processor that has an installed Lahey Fortran compiler and ESMF_COMPILER has been set to lahey, then the build system configuration file will be:

```
build_config/Linux.lahey.default/build_rules.mk
```

If you want to run a debug version of non-exhaustive unit tests, then you use these commands from /usr/local/esmf:

```
setenv ESMF_DIR /usr/local/esmf
make ESMF_BOPT=g ESMF_SITE=lahey ESMF_TESTEXHAUSTIVE=OFF tests_uni
```

If you are using ksh, then replace the setenv command with:

```
export ESMF_DIR=/usr/local/esmf
```

The results of the unit tests will be in:

```
/usr/local/esmf/test/testg/Linux.lahey.32.default/
```

At the end of unit test execution a script runs to analyze the results.

The script output indicates whether there are any unit test failures. If any unit tests fail, please check if the failures are listed as known bugs in the ESMF release page http://www.earthsystemmodeling.org/download/releases.shtml for your platform and compiler. If the failures are not listed please contact ESMF Support at esmf_support@ucar.edu Please indicate which unit tests are failing, and attach the output of the "make info" command to the email.

The script output indicates whether there are any unit test failures. The following is a sample from the script output:

```
The unit tests in the following files all pass:

src/Infrastructure/Array/tests/ESMF_ArrayUTest.F90
src/Infrastructure/ArrayDataMap/tests/ESMF_ArrayDataMapUTest.F90
src/Infrastructure/Base/tests/ESMF_BaseUTest.F90
src/Infrastructure/FieldBundle/tests/ESMF_FieldBundleUTest.F90
src/Infrastructure/FieldBundleDataMap/tests/ESMF_FieldBundleDataMapUTest.F90
src/Infrastructure/Config/tests/ESMF_ConfigUTest.F90
src/Infrastructure/DELayout/tests/ESMF_DELayoutUTest.F90
src/Infrastructure/Field/tests/ESMF_FRoute4UTest.F90
src/Infrastructure/Field/tests/ESMF_FieldUTest.F90
src/Infrastructure/FieldComm/tests/ESMF_FieldGatherUTest.F90
src/Infrastructure/FieldDataMap/tests/ESMF_FieldDataMapUTest.F90
src/Infrastructure/Grid/tests/ESMF_GridUTest.F90
src/Infrastructure/LocalArray/tests/ESMF_ArrayDataUTest.F90
src/Infrastructure/LocalArray/tests/ESMF_ArrayF90PtrUTest.F90
src/Infrastructure/LocalArray/tests/ESMF_LocalArrayUTest.F90
src/Infrastructure/LogErr/tests/ESMF_LogErrUTest.F90
src/Infrastructure/Regrid/tests/ESMF_Regrid1UTest.F90
src/Infrastructure/Regrid/tests/ESMF_RegridUTest.F90
src/Infrastructure/TimeMgr/tests/ESMF_AlarmUTest.F90
src/Infrastructure/TimeMgr/tests/ESMF_CalRangeUTest.F90
src/Infrastructure/TimeMgr/tests/ESMF_ClockUTest.F90
src/Infrastructure/TimeMgr/tests/ESMF_TimeIntervalUTest.F90
src/Infrastructure/TimeMgr/tests/ESMF_TimeUTest.F90
src/Infrastructure/VM/tests/ESMF_VMBarrierUTest.F90
src/Infrastructure/VM/tests/ESMF_VMBroadcastUTest.F90
src/Infrastructure/VM/tests/ESMF_VMGatherUTest.F90
src/Infrastructure/VM/tests/ESMF_VMScatterUTest.F90
src/Infrastructure/VM/tests/ESMF_VMSendVMRecvUTest.F90
src/Infrastructure/VM/tests/ESMF_VMUTest.F90
src/Superstructure/Component/tests/ESMF_CplCompCreateUTest.F90
src/Superstructure/Component/tests/ESMF_GridCompCreateUTest.F90
src/Superstructure/State/tests/ESMF_StateUTest.F90


The following unit test files failed to build, failed to execute or
crashed during execution:
```

```
src/Infrastructure/TimeMgr/tests/ESMF_CalendarUTest.F90
src/Infrastructure/VM/tests/ESMF_VMSendRecvUTest.F90


The following unit test files had failed unit tests:

src/Infrastructure/Field/tests/ESMF_FRoute8UTest.F90
src/Infrastructure/Grid/tests/ESMF_GridCreateUTest.F90


The following individual unit tests fail:

  FAIL  DELayout Get Test, ESMF_FRoute8UTest.F90, line 139
  FAIL  Grid Distribute Test, ESMF_GridCreateUTest.F90, line 198


The stdout files for the unit tests can be found at:
/home/bluedawn/svasquez/script_dirs/daily_builds/esmf/test/testO/ \
AIX.default.64.default

Found 1224 exhaustive multi processor unit tests, 1220 pass and 4 fail.
```

The following is an example of the output generated when a unit test fails:

```
ESMF_FieldUTest.stdout: FAIL  Unique default Field names Test, FLD1.5.1
& 1.7.1, ESMF_FieldUTest.F90, line 204  Field names
not unique
```

11.1.3 Running ESMF system tests

The system tests provided with the ESMF library evaluate:

- interface agreement between parts of the system
- behavior of the system as a whole

The current system test suite includes tests that perform layout reduction operations, redistribution-transpose, halo operations, component creation and intra-grid communication. Some of the system tests are no longer compatible with the current API, but are included in the release for completeness. A complete description of each available system test and its current compatibility status can be found at the ESMF website, http://www.earthsystemmodeling.org. The testing and validation page is accessible from the **Development** link on the navigation bar.

The following commands are used to build and run the system tests:

```
make [SYSTEM_TEST=xxx] system_tests
make [SYSTEM_TEST=xxx] system_tests_uni
```

The `system_tests_uni` target runs the tests on a single processor. The `system_tests` target runs the test on multiple processors.

If a particular SYSTEM_TEST is not specified, then all available system tests are built and run.

The results of the test can be found in the following location:

```
${ESMF_DIR}/test/test${ESMF_BOPT}/${ESMF_OS}.${ESMF_COMPILER}.${ESMF_ABI}. \
${ESMF_SITE}
```

For example, if your ESMF source files have been placed in your home directory:

```
~/esmf
```

and your platform and compiler configuration is:

```
Alpha multi-processor using the native compiler
```

and you want to run an optimized version of system test SimpleCoupling, then you use these commands from the directory ~/esmf.

```
setenv ESMF_PROJECT <project_name>
make ESMF_DIR=`pwd` SYSTEM_TEST=ESMF_SimpleCoupling system_tests
```

If you are using ksh then replace the setenv command with this:

```
export ESMF_PROJECT=<project_name>
```

The results will be in:

```
~/esmf/test/testO/OSF1.default.64.default/ESMF_SimpleCouplingSTest.stdout
```

At the end of system test execution a script runs to analyze the results.

The script output indicates whether there are any system test failures. If any system tests fail, please check if the failures are listed as known bugs in the ESMF release page http://www.earthsystemmodeling.org/download/releases.shtml for your platform and compiler. If the failures are not listed please contact ESMF Support at esmf_support@ucar.edu Please indicate which system tests are failing, and attach the output of the "make info" command to the email.

The script output indicates whether there are any system test failures. The following is a sample from the script output:

```
The following system tests passed:


src/system_tests/ESMF_CompCreate/ESMF_CompCreateSTest.F90
src/system_tests/ESMF_FieldExcl/ESMF_FieldExclSTest.F90
src/system_tests/ESMF_FieldHalo/ESMF_FieldHaloSTest.F90
src/system_tests/ESMF_FieldHaloPer/ESMF_FieldHaloPerSTest.F90
src/system_tests/ESMF_FieldRedist/ESMF_FieldRedistSTest.F90
src/system_tests/ESMF_FieldRegrid/ESMF_FieldRegridSTest.F90
```

```
src/system_tests/ESMF_FieldRegridMulti/ESMF_FieldRegridMultiSTest.F90
src/system_tests/ESMF_FieldRegridOrder/ESMF_FieldRegridOrderSTest.F90
src/system_tests/ESMF_FlowComp/ESMF_FlowCompSTest.F90
src/system_tests/ESMF_FlowWithCoupling/ESMF_FlowWithCouplingSTest.F90
src/system_tests/ESMF_SimpleCoupling/ESMF_SimpleCouplingSTest.F90
src/system_tests/ESMF_VectorStorage/ESMF_VectorStorageSTest.F90


The following system tests failed, did not build, or did not execute:


src/system_tests/ESMF_FieldRegridConserv/ESMF_FieldRegridConsrvSTest.F90
src/system_tests/ESMF_RowReduce/ESMF_RowReduceSTest.F90




The stdout files for the system_tests can be found at:
/home/bluedawn/svasquez/script_dirs/daily_builds/esmf/test/testO/ \
AIX.default.64.default

Found 14 system tests, 12 passed and 2 failed.
```

11.2 Running ESMF Examples

11.2.1 Example source code

Example source code for each class is found in the class's example directory. For example, source code for the Time Manager class examples are found in this directory:

```
ESMF_DIR/src/Infrastructure/TimeMgr/examples/
```

While the example code is formatted to be included in the documentation, it also runs and compiles to ensure accuracy. Examples generally contain simple usage of the basic methods for the class.

11.2.2 Building and running examples

The GNU makefile targets `examples` and `examples_uni` build and run programs found in a class's examples directory. After the examples are built, the `examples` target runs the examples using multiple processors, while `examples_uni` runs the examples on a single processor.

These targets first build the ESMF library.

Run from ESMF_DIR, this command will build and run all examples on multiple processors:

```
make examples
```

If the command is run in an example source code directory, then only the example from that directory will be built and run. The examples and output files are created in this directory:

```
ESMF_DIR/examples/examples$ESMF_BOPT/$ESMF_OS.$ESMF_COMPILER.$ESMF_ABI. \
$ESMF_SITE/
```

The name of an output file will begin with the name of the example that created it followed by .stdout.

At the end of examples execution a script runs to analyze the results.

The script output indicates whether there are any example failures. If any examples fail, please check if the failures are listed as known bugs in the ESMF release page http://www.earthsystemmodeling.org/download/releases.shtml for your platform and compiler. If the failures are not listed please contact ESMF Support at esmf_support@ucar.edu Please indicate which examples are failing, and attach the output of the "make info" command to the email.

The following is a sample from the script output:

```
The following examples passed:


src/Infrastructure/Array/examples/ESMF_ArrayCreateEx.F90
src/Infrastructure/Array/examples/ESMF_ArrayGetEx.F90
src/Infrastructure/ArrayComm/examples/ESMF_ArrayCommEx.F90
src/Infrastructure/ArrayDataMap/examples/ESMF_ArrayDataMapEx.F90
src/Infrastructure/ArraySpec/examples/ESMF_ArraySpecEx.F90
src/Infrastructure/FieldBundle/examples/ESMF_FieldBundleCreateEx.F90
src/Infrastructure/FieldBundleDataMap/examples/ESMF_FieldBundleDataMapEx.F90
src/Infrastructure/DELayout/examples/ESMF_DELayoutEx.F90
src/Infrastructure/Field/examples/ESMF_FieldCreateEx.F90
src/Infrastructure/Field/examples/ESMF_FieldFromUserEx.F90
src/Infrastructure/Field/examples/ESMF_FieldGlobalEx.F90
src/Infrastructure/Field/examples/ESMF_FieldWriteEx.F90
src/Infrastructure/FieldComm/examples/ESMF_FieldCommEx.F90
src/Infrastructure/FieldDataMap/examples/ESMF_FieldDataMapEx.F90
src/Infrastructure/LogErr/examples/ESMF_LogErrEx.F90
src/Infrastructure/Regrid/examples/ESMF_RegridEx.F90
src/Infrastructure/Route/examples/ESMF_RouteEx.F90
src/Infrastructure/TimeMgr/examples/ESMF_AlarmEx.F90
src/Infrastructure/TimeMgr/examples/ESMF_CalendarEx.F90
src/Infrastructure/TimeMgr/examples/ESMF_ClockEx.F90
src/Infrastructure/TimeMgr/examples/ESMF_TimeEx.F90
src/Infrastructure/VM/examples/ESMF_VMAllFullReduceEx.F90
src/Infrastructure/VM/examples/ESMF_VMComponentEx.F90
src/Infrastructure/VM/examples/ESMF_VMDefaultBasicsEx.F90
src/Infrastructure/VM/examples/ESMF_VMGetMPICommunicatorEx.F90
src/Infrastructure/VM/examples/ESMF_VMScatterVMGatherEx.F90
src/Infrastructure/VM/examples/ESMF_VMSendVMRecvEx.F90
src/Superstructure/Component/examples/ESMF_AppMainEx.F90
src/Superstructure/Component/examples/ESMF_CplEx.F90
```

```
src/Superstructure/Component/examples/ESMF_GCompEx.F90
src/Superstructure/State/examples/ESMF_StateEx.F90
src/Superstructure/State/examples/ESMF_StateReconcileEx.F90


The following examples failed, did not build, or did not execute:


src/Infrastructure/Grid/examples/ESMF_GridCreateEx.F90
src/Infrastructure/TimeMgr/examples/ESMF_TimeIntervalEx.F90


The stdout files for the examples can be found at:
/home/bluedawn/svasquez/script_dirs/daily_builds/esmf/examples/
examplesO/AIX.default.64.default


Found 34 examples, 32 passed and 2 failed.
```

11.3 Validating an existing ESMF installation

It is becoming increasingly common to find pre-installed ESMF libraries on professionally maintained HPC systems. Often multiple versions of ESMF are available via environment modules. Before using such a third-party ESMF installation, a user may want ot validate that it is working correctly. System administrators also often need a simple method to re-validate an existing ESMF installation, e.g. after a system update. ESMF offers a simple way to build and run the full regression suite against an existing installation.

A second ESMF source tree is used to run full regression tests against an existing ESMF installation. To support this, the second source tree must be of the exact same version as the ESMF installation to be tested. The two critical environment variables used are `ESMF_TESTESMFMKFILE`, and `ESMFMKFILE`. The following bullets outline the procedure:

- Check out the same version of ESMF as the installation to be validated.
- Change into the root directory of the checked out directory tree, and set the `ESMF_DIR` environment variable to the current working directory.
- Set the `ESMFMKFILE` environment variable to point to the `esmf.mk` file of the installation to be validated. If the ESMF installation is available via the `module` command, `ESMFMKFILE` will typically be set when loading the module.
- Set the `ESMF_TESTESMFMKFILE` environment variable to `ON`.
- Set the `ESMF_COMPILER`, `ESMF_COMM`, and `ESMF_BOPT` environment variables to match the values from the `esmf.mk` file.
- Make sure the build environment is set up properly to match the ESMF installation to be validated. On systems using the `module` command, this means loading the correct modules.

At this point all of the test targets discussed in sections 11.1 and 11.2 are available. The build targets use the test and example sources under the local (secondary) source tree, but compile and link against the ESMF library pointed to by `ESMFMKFILE`. A fully functional installation is expected to pass all regression tests.

12 Architectural Overview

The ESMF architecture is characterized by the layering strategy shown in Figure 1. User code components that implement the *science* portions of an application, for example a sea ice or land model, are sandwiched between two layers. The upper layer is denoted the **superstructure** layer and the lower layer the **infrastructure** layer. The role of the superstructure layer is to provide a shell which encompasses user code and provides a context for interconnecting input and output data streams between components. The key elements of the superstructure are described in Section 12.2. These elements include classes that wrap user code, ensuring that all components present consistent interfaces. The infrastructure layer provides a foundation that developers of science components can use to speed construction and to ensure consistent, guaranteed behavior. The elements of the infrastructure include constructs to support parallel processing with data types tailored to Earth science applications, specialized libraries to support consistent time and calendar management and performance, error handling and scalable I/O tools. The infrastructure layer is described in Section 12.3. A hierarchical combination of superstructure, user code components, and infrastructure are joined together to form an ESMF application.

12.1 Key Concepts

The ESMF architecture and programming paradigm are based upon five key concepts: modularity, flexibility, hierarchical organization, communication within components, and a uniform communication API.

12.1.1 Modularity

The ESMF design is based upon modular Components. There are two types of Components, one of which represents models (Gridded Components) and one which represents couplers (Coupler Components). Data are always passed between Components using a data structure called a State, which can store Fields, FieldBundles of Fields, Arrays, and other States. A Gridded Component stores no information about the internals of the Gridded Components that it interacts with; this information is passed in through the argument lists of the initialize, run, and finalize methods. The information that is passed in through the argument list can be a State from another Gridded Component, or it can be a function pointer that performs a computation or communication on a State. These function pointers are called Transforms, and they are available as AttachableMethods created by Coupler Components. They are called inside the Gridded Component they are passed into. Although Transforms add some complexity to the framework (and their use is not required), they are what will enable ESMF to accommodate virtually any model of communication between Components.

Modularity means that an ESMF component stores nothing about the internals of other components. This allows components to be used more easily in multiple contexts.

12.1.2 Flexibility

The ESMF does not dictate how models should be coupled; it simply provides tools for creating couplers. For example, both a hub-and-spokes type coupling strategy and pairwise strategies are supported. The ESMF also allows model communications to occur mid-timestep, if desired. Sequential, concurrent, and mixed modes of execution are supported.

The ESMF does not impose restrictions on how data flows through an application. This accommodates scientific innovation - if you want your atmospheric model to communicate with your sea ice model mid-timestep, ESMF will not stop you.

12.1.3 Hierarchical organization

ESMF allows applications to be composed hierarchically. For example, physics and dynamics modules can be defined as separate Gridded Components, coupled together with a Coupler Component, and all of these nested within a single atmospheric Gridded Component. The atmospheric Gridded Component can be run standalone, or can be included in a larger climate or data assimilation application. See Figure 2 for an illustrative example.

The data structure that enables scalability in ESMF is the derived type Gridded Component. Fortran alone does not allow you to create generic components - you'd have to create derived types for PhysComp, and DynComp, and PhysDynCouplerComp, and AtmComp. In ESMF, these are always of type GridComp or CplComp, so they can be called by the same drivers (whether that driver is a standard ESMF driver or another model), and use the same methods without having to overload them with many specific derived types. It is the same idea when you want to support different implementations of the same component, like multiple dynamics.

The ESMF defines a hierarchical, scalable architecture that is natural for organizing very complex applications, and for allowing exchangeable Components.

12.1.4 Communication within Components

Communication in ESMF always occurs within a Component. It can occur internal to a Gridded Component, and have nothing to do with interactions with other Components (setting aside synchronization issues), or it can occur within a Coupler Component or a transform generated by a Coupler Component. A result of the rule that all communication happens within a Component is that Coupler Components must always be defined on the union of all the Components that they couple together. Models can choose to use whatever mechanism they want for intra-model communications.

The point is that although the ESMF defines some simple rules for communication, the communication mechanism that the framework uses is not hardwired into its architecture - the sends and receives or puts and gets are enclosed within Gridded Components, Coupler Components and Transforms. The intent is to accommodate multiple models of communication and technical innovations.

12.1.5 Uniform communication API

ESMF has a single API for shared and distributed memory that, unlike MPI, accounts for NUMA architectures and does not treat all processes as being identical. It is possible for users to set ESMF communications to a strictly message passing mode and put in their own OpenMP commands.

The goal is to create a programming paradigm that is performance sensitive to the architecture beneath it without being discouragingly complicated.

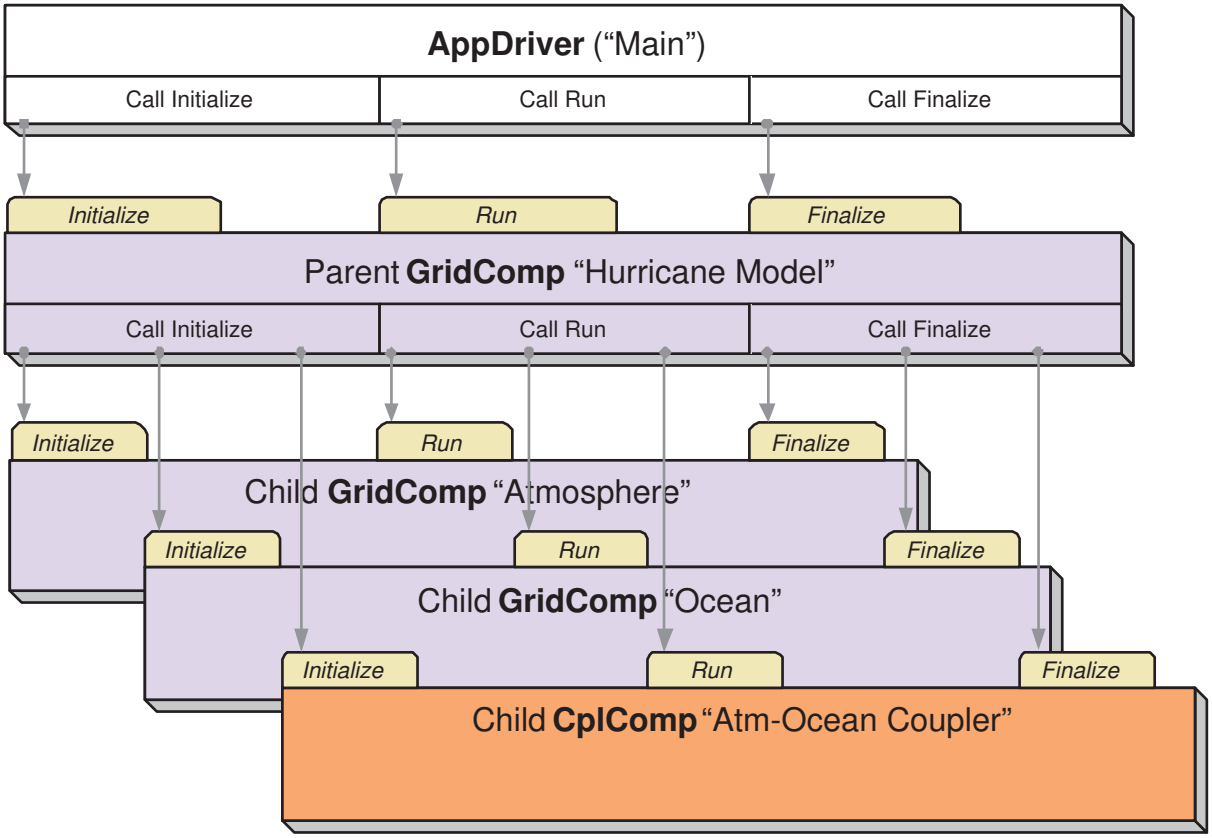
12.2 Superstructure

The ESMF superstructure layer in a unifying context within which user components are interconnected. Classes called **Gridded Components**, **Coupler Components**, and **States** are used within the superstructure to achieve this flexibility.

12.2.1 Import and export State classes

User code components under ESMF use special interface objects for Component to Component data exchanges. These objects are of type import State and export State. These special types support a variety of methods that allow user code components to do things like fill an export State object with data to be shared with other components or query an

Figure 2: A typical building block for an ESMF application consists of a parent Gridded Component, two or more child Gridded Components, and a Coupler Component. The parent Gridded Component is called by an application driver. All ESMF Components have initialize, run, and finalize methods. The diagram shows that when the application driver calls initialize on a parent Gridded Component, the call cascades down to all of its children, so that the result is that the entire "tree" of Components is initialized. The run and finalize methods work the same way. In this example a hurricane simulation is built from ocean and atmosphere Gridded Components. The data exchange between the ocean and atmosphere is handled by an ocean-atmosphere Coupler Component. Since the whole hurricane simulation is a Gridded Component, it could be easily be treated as a child and coupled to another Gridded Component, rather than being driven directly by the application driver. A similar diagram could be drawn for an atmospheric model containing physics and dynamics components, as described in Section 12.1.3.



import State object to determine its contents. In keeping with the overall requirements for high-performance it is permitted for import State and export State contents to use references or pointers to Component data, so that costly data copies of potentially large data structures can be avoided where possible. The content of an import State and an export State can be made self-describing.

12.2.2 Interface standards

The import State and export State abstractions are designed to be flexible enough so that ESMF does not need to mandate a single format for fields. For example, ESMF does not prescribe the units of quantities exported or imported. However, ESMF does provide mechanisms to describe units, memory layout, and grid coordinates. This allows the ESMF software to support a range of different policies for physical fields. The interoperability experiments that we are using to demonstrate ESMF make use of the emerging CF conventions [1] for describing physical fields. This is a policy choice for that set of experiments. The ESMF software itself can support arbitrary conventions for labeling and characterizing the contents of States.

12.2.3 Gridded Component class

The Gridded Component class describes a user component that takes in one import State and produces one export State. Examples of Gridded Components are major Earth system model components such as land surface models, ocean models, atmospheric models and sea ice models. Components used for linear algebra manipulations in a state estimation or data assimilation optimization procedure are also created as Gridded Components. In general the fields within an import State and export State of a Gridded Component will use the same discrete grid.

12.2.4 Coupler Component class

The other top-level Component class supported in the ESMF architecture is a Coupler Component. This class is used for Components that take one or more import States as input and map them through spatial and temporal interpolation or extrapolation onto one or more output export States. In a Coupler Component it is often the case that the export State(s) is on a different discrete grid to that of the import State(s). For example, in a coupled ocean-atmosphere simulation a Coupler Component might be used to map a set of sea-surface fields in an ocean model to appropriate planetary boundary layer fields in an atmospheric model.

12.2.5 Flexible data and control flow

Import States, export States, Gridded Components and Coupler Components can be arrayed flexibly within a superstructure layer. Using these constructs, it is possible to configure a set of Components with multiple pairwise Coupler Components, Figure 4. It is also possible to configure a set of concurrently executing Gridded Components joined through a single Coupler Component of the style shown in Figure 3.

The set of superstructure abstractions allows flexible data flow and control between components. However, components will often use different discrete grids, and time-stepping components may march forward with different time intervals. In a parallel compute environment different components may be distributed in a different manner on the underlying compute resources. The ESMF infrastructure layer provides elements to manage this complexity.

Figure 3: ESMF supports configurations with a single central Coupler Component. In this case inputs from all Gridded Components are transferred and regridded through the central coupler.

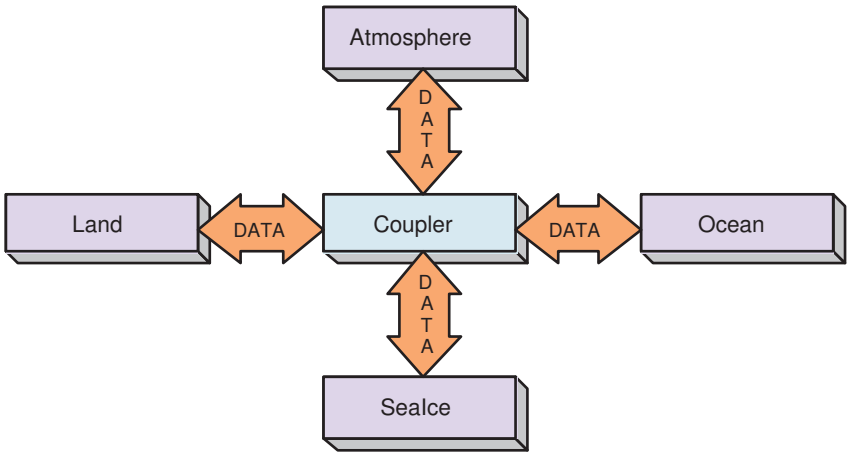


Figure 4: ESMF also supports configurations with multiple point to point Coupler Components. These take inputs from one Gridded Component and transfer and regrid the data before passing it to another Gridded Component. This schematic shows a flow of data between two Coupler Components that connect three Gridded Components: an atmosphere model with a land model, and the same atmosphere model with a data assimilation system.

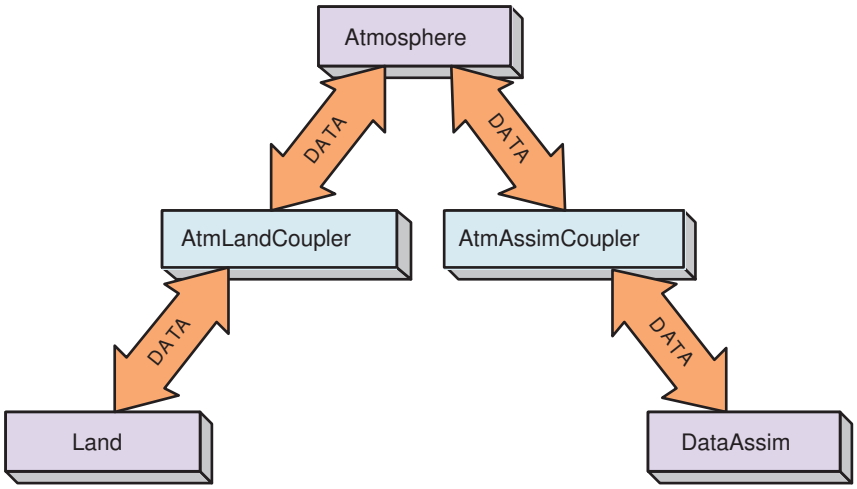
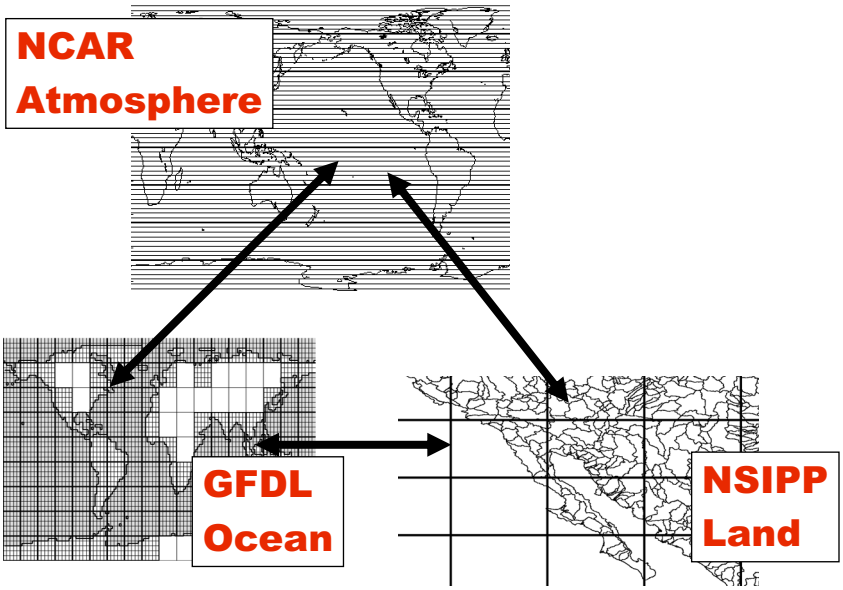


Figure 5: Schematic showing the coupling of components that use different discrete Grids and different time-stepping. In this example, Component *NCAR Atmosphere* might use a spectral Grid based on spherical harmonics, Component *GFDL Ocean* might use a latitude-longitude Grid but with a patched decomposition that does not include land masses, and Component *NSIPP Land* might use a m osaic-based Grid for representing vegetation patchiness and a catchment area based Grid for river routings. The ESMF infrastructure layer contains tools to help develop software for coupling between Components on different Grids, mapping between Components with different distributions in a multi-processor compute environment and synchronizing events between Components with different time-stepping intervals and algorithms.



12.3 Infrastructure

Figure 5 illustrates three Gridded Components, each with a different Grids, being coupled together. In order to achieve this coupling several steps beyond defining import State and export State objects to act as data conduits are required. Coupler Components are needed that can interpolate between the different Grids. The necessary transformations may also involve mapping between different units and/or memory layout conventions for the Fields that pass between Components. In a parallel compute environment the Coupler Components may also be required to map between different domain decompositions. In order to advance in time correctly the separate Gridded Components must have compatible notions of time. Approaches to parallelism within the Gridded Components must also be compatible. The **Infrastructure** layer contains a set of classes that address these issues and assist in managing overall system complexity.

12.3.1 FieldBundle, Field and Array classes

FieldBundle, Field and Array classes contain data together with descriptive physical and computational attribute information. The physical attributes include information that describes the units of the data. The computational attributes include information on the layout in memory of the field data. The Field class is primarily geared toward structured data. A comparable class, called Location Stream, provides a self-describing container for unstructured observational data streams.

12.3.2 Grid class

The *Grid* class is an extensible class that holds discrete grid information. It has subtypes that allow it to serve as a container for the full range of different physical grids that might arise in a coupled system. In the example in figure 5 objects of type Grid would hold grid information for each of the spectral grid, the latitude-longitude grid, the mosaic grid and the catchment grid.

The Grid class is also used to represent the decomposition of a data structure into subdomains, typically for parallel processing purposes. The class is designed to support a generalized "ghosting" for tiled decompositions of finite difference, finite volume and finite element codes.

12.3.3 Time and Calendar management

To support synchronization between Components, several time and calendar management classes are provided. These capabilities are provided in the Time, Time Interval, Calendar, Clock, and Alarm classes. These classes allow Gridded and Coupler Component processing to be latched to a common controlling Clock, and to schedule notification of regular events, such as a coupling intervals, and unique events.

12.3.4 Config resource file manager

The Config class is a utility for accessing configuration files that are in ASCII format. This utility enables configuration files to be prepared using more flexible formatting than Fortran namelists - for example, it permits the input of tables of data.

12.3.5 DELayout and virtual machine

To provide a mechanism for ensuring performance portability, ESMF defines DELayout and virtual machine (VM) classes. These classes provide a set of high-level and platform independent interfaces to performance critical parallel processing communication routines. These routines can be tuned to specific platforms to ensure optimal parallel performance on many platforms.

12.3.6 Logging and error handling

The LogErr class is designed to aid in managing the complexity of multi-Component applications. It provides ESMF with a unified mechanism for managing logs and error reporting.

12.3.7 File input and output

The infrastructure layer will define a set of *IO* classes for storing and retrieving Array, Field, and Grid information to and from persistent storage.

13 How to Adapt Applications for ESMF

In this section we describe how to bring existing applications into the framework.

13.1 Individual Components

- Decide what parts will become Gridded Components

  A Gridded Component is a self-contained piece of code which will be initialized, will be called once or many times to run, and then will be finalized. It will be expected to either take in data from other components/models, produce data, or both.

  Generally a computational model like an ocean or atmosphere model will map either to a single component or to a set of multiple nested components.

- Decide what data is produced

  A component provides data to other components using an ESMF State object. A component should fill the State object with a description of all possible values that it can export. Generally, a piece of code external to the component (the AppDriver, or a parent component) will be responsible for marking which of these items are actually going to be needed. Then the component can choose to either produce all possible data items (simpler but less efficient) or only produce the data items marked as being needed. The component should consult the CF data naming conventions when it is listing what data it can produce.

- Decide what data is needed

  A component gets data from other components using an ESMF State object. The application developer must figure out how to get any required fields from other components in the application.

- Make the data blocks private

  A component should communicate to other components only through the framework. All global data items should be private to Fortran modules, and ideally should be isolated to a single derived type which is allocated at run time.

- Divide the code up into start/middle/end phases

  A component needs to provide 3 routines which handle initialization, running, and finalization. (For codes which have multiple phases of initialize, run, and finalize it is possible to have multiple initialize, run, and finalize routines.)

  The initialize routine needs to allocate space, initialize data items, boundary conditions, and do whatever else is necessary in order to prepare the component to run.

  For a sequential application in which all components are on the same set of processors, the run phase will be called multiple times. Each time the model is expected to take in any new data from other models, do its computation, and produce data needed by other components. A concurrent model, in which different components are run on different processors, may execute the same way. Alternatively, it may have its run routine called only once and may use different parts of the framework to arrange data exchange with other models. This feature is not yet implemented in ESMF.

  The finalize routine needs to release space, write out results, close open files, and generally close down the computation gracefully.

- Make a "Set Services" subroutine

  Components need to provide only a single externally visible entry point. It will be called at start time, and its job is to register with the framework which routines satisfy the initialize, run, and finalize requirements. If it has a single derived type that holds its private data, that can be registered too.

- Create ESMF Fields and FieldBundles for holding data

  An ESMF State object is fundamentally an annotated list of other ESMF items, most often expected to be ESMF FieldBundles (groups of Fields on the same grid). Other things which can be placed in a State object are Fields, Arrays (raw data with no gridding/coordinate information) and other States (generally used by coupling

code). Any data which is going to be received from other components or sent to other components needs to be represented as an ESMF object.

To create an ESMF Field the code must create an ESMF Array object to contain the data values, and usually an ESMF Grid object to describe the computational grid where the values are located. If this is an observational data stream the locations of the data values will be held in an ESMF Location Stream object instead of a Grid.

- Be able to read an ESMF clock

  During the execution of the run routine, information about time is transferred between components through ESMF Clocks. The component needs to be able to at least query a Clock for the current time using framework methods.

- Decide how much of the lower level infrastructure to use

  The ESMF framework provides a rich set of time management functions, data management and query functions, and other utility routines which help to insulate the user's code from the differences in hardware architectures, system software, and runtime environments. It is up to the user to select which parts of these functions they choose to use.

13.2 Full Application

- Decide on which components to use

  Select from the set of ESMF components available.

- Understand the data flow in order to customize a Coupler Component

  Examine what data is produced by each component and what data is needed by each component. The role of Coupler Components in the ESMF is to set up any necessary regridding and data conversions to match output data from one component to input data in another.

- Write or adapt a Coupler Component

  Decide on a strategy for how to do the coupling. There can be a single coupler for the application or multiple couplers. Single couplers follow a "hub and spoke" model. Multiple couplers can couple between subsets of the components, and can be written to couple either only one-way (e.g. output of component A into input of component B), or two-way (both A to B and B to A).

  The coupler must understand States, Fields, FieldBundles, Grids, and Arrays and ESMF execution/environment objects such as DELayouts.

- Use or adapt a main program

  The main program can be a copy of a driver found in any of the `system_tests` sub-directories. The customization needed is to `use` the correct Component module files, to gain access to the `SetServices` routines.

  Although ESMF provides example source code for the main program, it is **not** considered part of the framework and can be changed by the user as needed.

  The final thing the main program must do is call `ESMF_Finalize()`. This will close down the framework and release any associated resources.

  The main program is responsible for creating a top-level Gridded Component, which in turn creates other Gridded and Coupler Components. We encourage this hierarchical design because it aids in extensibility - the top level Gridded Component can be nested in another larger application. The top-level component contains the main time loop and is responsible for calling the `SetServices` entry point for each child component it creates.

14 Glossary

This glossary defines terms used in Earth system modeling to describe parallel computer architectures, grids and grid decompositions, and numerical and computational methods.

360-day calendar A calendar in which every one of twelve months has thirty days. See also Calendar, no-leap calendar.

Accumulator A facility for collecting and averaging data values. Generally accumulators are associated with temporal averaging, although they might be associated with other weighted averaging operations. ESMF does not yet have accumulators.

Application Programming Interface (API) API refers to the set of routines and types in a software package that are available to its users. It doesn't include private or internal routines or types.

Alarm Like a real alarm clock, the ESMF Alarm class notifies the user of an event that occurs at a particular time (or set of times). In order to determine whether it is "ringing", an ESMF Alarm is "read" by an explicit application action. An Alarm is associated with a particular Clock.

Application A coherent computational entity run as a single executable or set of communicating executables. It typically consists of a set of interacting components. See also component.

Array An ESMF class that represents a multi-dimensional data array. Unlike a native Fortran or C++ array, an ESMF Array can store information about halo points. See also halo.

Background grid A background grid associates each point in an observational data stream (Location Stream) with a location on a grid. A single grid cell may contain zero or more Location Stream points. See also Location Stream, cell.

BUFR Binary Universal Form of Representation. This is a World Meteorological Organization data format. See BUFR links.

FieldBundle The ESMF FieldBundle class represents a set of fields that are associated with the same physical grid and are distributed in the same fashion across the same physical axes. Fields within a FieldBundle may be staggered differently and may have different (non-distributed) dimensions. See also Field, Packed FieldBundle, Loose FieldBundle.

Calendar The Calendar is an ESMF class that stores a representation of a particular calendar type, such as Gregorian. See also specific calendar types such as 360-day and no-leap.

Cell A physical location that is specified by both its extent (vertices) and nominal central location, and is associated with a single integer index value or a set of integer index values (e.g. (i) for 1-d, (i,j) for 2-d, (i,j,k) for 3d). See also index.

CF Conventions Climate and Forecast Conventions. These are emerging conventions for expressing Earth science metadata. See the CF home page.

Change Review Board (CRB) The Change Review Board is the ESMF management body that sets project schedules and priorities. Its Terms of Reference are in the ESMF Project Plan.

Clock Clock is an ESMF class that tracks the passage of time and reports the current time instant. An ESMF Clock is stepped forward in increments of a time step, and can be associated with one or more Alarms. See also Time, Time Interval, Alarm.

Component The ESMF Component class represents large-scale computational entities associated with a particular physical process or computational function, such as a land model. Currently ESMF supports Gridded Component and Coupler Component classes. Components may be generic or user-supplied.

Computational domain For a given DE, the data points that have unique global indices and are updated by the DE. See also exclusive domain, total domain, halo.

Computational resource Something that appears as a physical or virtual computer resource. Example of computational resources are a CPU, a network connection, a communication API, a protocol, a particular network fabric or a piece of computer memory.

Concurrent execution Concurrent execution of model components occurs when two or more components, whether in the same or different executables, run simultaneously. See also Sequential execution.

Congruent If all Fields in a FieldBundle contain the same data type, rank, shape, and relative locations, the FieldBundle is said to be congruent.

Coupler Component An ESMF Component that includes all data and actions needed to enable communication between two or more Gridded Components. See also component, Gridded Component.

Curvilinear grid A curvilinear grid is a logically rectangular grid in which coordinates in physical space must be specified by giving the explicit coordinates for each point. Curvilinear grids are often uniform or rectilinear grids that have been warped, for example in order to place a pole over land points so it does not affect the computations performed on an ocean model grid. See also logically rectangular grid, Uniform grid, Rectilinear grid.

Data dependency The property of a computational operator that defines the data indices required to perform the computation at a point.

Data parallel The quality of an application that allows roughly the same calculation to be performed by all processors at the same time on the same data set, which is partitioned among multiple memory locations. Single components that do not contain nested components are often data parallel. See also task parallel, SPMD, MPMD.

Data transpose Rearrangement of data arrays that share the same global domain.

Day of year The day number in the calendar year. January 1 is day 1 of the year. Day of year expressed in a floating point format is used to express the day number plus the time of day. For example, assuming a Gregorian calendar:

date	day of year
10 January 2000, 6Z	10.25
31 December 2000, 18Z	366.75

DE Short for Decomposition Element.

DELayout DELayout is the ESMF class that defines the topology of a set of DEs and specifies how the DEs are assigned to PETs in an ESMF Virtual Machine.

Decomposition Element (DE) A DE is the smallest unit of decomposition of a computational task. DEs are virtual units, not necessarily having a 1-to-1 correspondence to the Persistent Execution Threads (PETs) of a VM or the physical Processing Elements (PEs) in the underlying physical machine. Consequently there are no restrictions on the number of DEs that can be created. The application writer may chose the number of DEs to best match the computational problem and the employed algorithm. A DELayout assigns a topology to Decomposition Elements. See also DELayout.

Deep object In an environment in which the calling and implementation language of a library are different, deep objects are defined as those whose memory is allocated by the implementation language. See also shallow object.

Distributed Grid DistGrid is the ESMF class that defines the decomposition of a Grid's global index space across a DELayout. DistGrid objects are contained in an ESMF Grid. See also Grid, DELayout.

Distribution The function that expresses the relationship between the indices in a Distributed Grid and the elements in a DELayout. See also Distributed Grid, DELayout.

Domain decomposition The act of grid distribution: creating a DistGrid, and associating grid points with the DistGrid. The dimensionality of the domain decomposition is the same as the dimensionality of the associated DistGrid.

Exact The word exact is used to denote entities, such as time instants and time intervals, for which truncation-free arithmetic is required.

Exchange grid A grid whose vertices are formed by the intersection of the vertices of two overlying grids. Each cell in the exchange grid overlies exactly one cell in each grid of the exchange. See also grid, cell.

Exchange Packets Exchange Packets are a private ESMF class that contains data in an optimal form for data transfers.

Exclusive domain For a given DE, the set of data points that are not replicated on any other DE. See also total domain, computational domain, halo.

Executable A program that is under independent control by the operating system.

Export State The data and metadata that a component can make available for exchange with other components. This may be data at a physical boundary (e.g land-atmosphere interface) or in other cases, it might be the entire model state. See also State, import State.

Field The ESMF Field class represents a tangible or derived quantity defined within a continuous region of space. The Field class includes the physical grid associated with the quantity and a decomposition that specifies how data associated with points in the physical grid are distributed in computer memory and/or how computational work is divided among threads. A Field also includes a specification of gridpoint staggering and any metadata necessary for a full description of its data. See also Grid.

Framework We use the term framework to refer to a structured collection of software building blocks that can be used and customized to develop components, assemble them into an application, and run the application.

Generic component A generic component is one supplied by the framework. The user is not expected to customize or otherwise modify it. ESMF does not currently contain any generic components. See also user component, component.

Generic transform A generic transform is an operation supplied by the framework, for example, a method that converts gridded data from one supported grid and/or decomposition to another using a specified technique. See also user transform.

Global domain A global domain refers to the full extent of a DELayout or Grid.

Global reduction Reduction operations (sum, max, min, etc.) that condense data distributed over a global domain. See also global broadcast.

Global broadcast Scatter operations on data distributed over a global domain. See also global reduction.

Gregorian The Gregorian calendar is the most widely used calendar in the world. The calendar's zeroth year is at the birth of Jesus Christ. Years after the origin (anno Domini, or AD) are positive, and before (Before Christ, or BC) are negative. Several corrections (leap year, 100 year, 400 year) are necessary to keep the calendar aligned with solar cycles. See also Calendar.

GRIB The GRid in Binary Data format from the World Meteorological Organization. This format is frequently used by operational weather centers. See the GRIB home page, and GRIB2 reference guide.

Grid The discrete division of space associated with a particular coordinate system. The ESMF Grid class contains coordinate, domain decomposition, and memory organization information required to manipulate Fields, as well as to create and execute Grid transforms. See also Distributed Grid, DELayout.

Grid staggering A descriptor of relative locations of scalar and vector data on a structured grid. On different staggered grids, vector data may lie at cell faces or vertices, while scalar data may lie in the interior.

Grid topology Description of data connectivities for a grid.

Grid union The formation of a new grid by taking the union of the vertices of two input grids. See also Grid.

Gridded Component An ESMF class that represents a component that is associated with one or more grids. No requirements may be placed on the physical content of a Gridded Component's data or on the nature of its computations. See also component, Coupler Component.

Halo For a given DE, a halo is a set of data points from the computational domains of neighboring DEs that are replicated locally for computational convenience. A halo can be defined as all the data points in a DE's total domain excluding those in its computational domain. See also computational domain, total domain, exclusive domain.

Halo update A halo update operation involves synchronization of the values of some or all halo points with the current values of those points on other DEs. See also halo.

Import State The data and metadata that a component requires from other components in order to run. See also State, export State.

Index An integer value associated with a set of coordinates.

Index space The space implied by a set of indices. An index space has a defined dimensionality and connectivity.

Index space location A location within an index space. An index space location may be fractional. See also physical location.

Instantiate To create an actual instance of a software class. For example, each variable of derived type Field in an ESMF Fortran application is an instance of the Field class.

Interface Used generally to refer to a set of operations that characterize the behavior of a class or a component. Also used to refer to the name and argument list of a particular method.

Joint Milestone Codeset(JMC) Joint Milestone Codeset. This is the set of climate, weather and data assimilation applications used as ESMF testbeds during the initial NASA-funded phase of ESMF development.

Joint Specification Team(JST) The JST is the body of developers and users who collaborate to create the ESMF software. The main form of communication for the JST is the weekly telecon. Terms of Reference are in the ESMF Project Plan.

LocalArray A LocalArray is the portion of an ESMF Array that resides on a particular DE. See also Array.

LocalTile A LocalTile is the portion of a grid Tile that resides on a particular DE. See also Tile.

Location Stream An ESMF class that represents a list of locations with no assumed relationship between these locations. The elements of a Location Stream are not assumed to share the same metadata. Location Streams are not yet implemented. See also background grid.

Logically rectangular grid A grid in which a set of coordinates (x,y,z, ...) in physical space can be mapped one-to-one to a set of regularly spaced points (i,j,k, ...) in a rectangular logical space, preserving proximate relationships. See also Grid.

Loose FieldBundle A loose FieldBundle is an ESMF FieldBundle object that contains fields whose data is not contiguous in memory. See also FieldBundle, packed FieldBundle.

Machine model A generic representation of the computing platform architecture.

Mask A data field marking a span within a larger data field.

Memory domain The portion of memory associated with the data on a given DE. The memory domain is always at least as large as the total domain. See also total domain.

Mosaic grid A mosaic grid is composed of multiple logically rectangular grid tiles that are connected at their edges, for example, a cubed sphere grid. See also grid tile.

MPMD Multiple Program Multiple Datastream. Multiple executables, any of which could itself be an SPMD executable, executing independently within an application. See also SPMD.

Namelist An I/O feature supported by Fortran that defines a structured syntax for creating text files of initial variable settings and defines language features for compactly reading the files. The syntax for Namelist files can be found in most Fortran manuals and tutorial texts.

NetCDF Network Common Data Form. This is a popular I/O library and data format in the Earth sciences. See NetCDF home page.

Node A node is a set of computational resources that is typically located in close proximity on a computing platform and that is associated with a single shared memory buffer.

No-leap calendar In this calendar every year uses the same months and days per month as in a non-leap year of a Gregorian calendar. See also Calendar, 360-day calendar.

Packed FieldBundle A packed FieldBundle is an ESMF FieldBundle object that contains a data buffer with field data arranged contiguously in memory. See also FieldBundle, loose FieldBundle.

Parallel execution The term parallel execution refers to the execution of a software application on more than one PE. See also serial.

PE Short for Processing Element.

PET Short for Persistent Execution Thread.

Persistent Execution Thread (PET) Provides a path for executing an instruction sequence. A PET has a lifetime at least as long as the associated data objects. The PET is a key abstraction used in the ESMF Virtual Machine.

Physical location A point in physical space to which a data point pertains. See also index space location.

Platform The processor hardware, operating system, compiler and parallel library that together form a unique compilation target.

Processing Element (PE) A Processing Element (PE) is the smallest physical processing unit available on a particular hardware platform.

Rectilinear grid A rectilinear grid is a logically rectangular grid in which the coordinates in physical space can be fully specified by the spacing of grid points along each grid axis. The gridpoints are located where the coordinate values intersect. The spacing along each axis may vary. See also logically rectangular grid, Uniform grid, Curvilinear grid.

Regrid weight generation methods and applications The collective term for all ESMF interfaces that compute regridding weights. This covers Fortran methods: `ESMF_FieldRegridStore()`, `ESMF_FieldBundleRegridStore()`, `ESMF_RegridWeightGen()`, and the command line application: `ESMF_RegridWeightGen`.

Scheduler An operating system component that assigns system resources (processors, memory, CPU time, I/O channels, etc.) to executables.

Search `Search` refers to the process of determining which processors must exchange data (and how much) when regridding between decomposed grids. See also sweep.

Sequential execution Sequential execution of model components describes the case in which one component waits for another to finish before it begins to run. Components executing sequentially may be in the same or different executables and may have coincident or non-overlapping memory distributions. See Concurrent execution.

Serial Execution The term serial execution refers to the execution of a software application on only one PET. See also parallel execution.

Shallow object In an environment in which the calling and implementation language of a library are different, shallow objects are defined as those whose memory is allocated by the calling language. See also deep object.

Span The physical extent associated with a grid.

SPMD Single Program Multiple Datastream. A single executable, possibly with many components (representing for example the atmosphere, the ocean, land surface) executing serially or concurrently. See also MPMD.

State The ESMF State class may contain Arrays, FieldBundles, Fields, or other States. It is used to transfer data between components. See also import State, export State.

Sweep `Sweep` refers to the regridding process of looping through lists of cells from one grid, hunting for interactions with a specified point or subsegment from the other grid. The type of interaction depends on the regrid method and is either an intersection with an identified subsegment or containment of an identified point. The limitation of the range of cells that must be examined is also considered part of the sweep algorithm. See also search.

System time Time spent doing system tasks such as I/O or in system calls. May also include time spent running other processes on a multiprocessor system. See also user time, wall clock time.

Task parallel The quality of an application that allows different calculations to be performed by different processors at the same time on what are usually different data sets. Large-scale task parallelism is often associated with multi-component applications in which each component represents a separate domain or function. Task parallel applications may be run with components executing either sequentially or concurrently, and either in a SPMD or MPMD mode. See also data parallel, SPMD, MPMD, sequential execution, concurrent execution.

Some grids used in Earth system modeling, such as cubed sphere grids, are most naturally represented as a set of logically rectangular grids that are connected at their edges. Following V. Balaji [2006] we refer to each of the logically rectangular grids in a composite grid, or mosaic grid, as a Tile. See also mosaic grid, LocalTile.

Time Time is an ESMF class that is made up of a time and date and an associated calendar. It may include a time zone. *Jan 3rd 1999, 03:30:24.56s, UTC* is one example of a Time. See also Calendar.

Time Interval Time Interval is an ESMF class that represents the period between any two time instants, measured in units, such as days, seconds, and fractions of a second. The periods *2 days and 10 seconds*, *86400 and 1/3 seconds* and *31104000.75 seconds* are all possible values for Time Intervals. Mathematical operations such as addition, multiplication, and subdivision can be applied to Time Intervals, and they can have negative values. See also Time

Total domain For a given DE, the entirety of the data points allocated, included replicated points from neighboring DEs. See also computational domain, exclusive domain, halo

A logically rectangular grid in which the coordinates in physical space can be completely specified by the two sets of coordinates that define the opposing corner points of the physical span. The coordinates of each point in physical space can be obtained by interpolating from the corner points, using the evenly spaced logical grid to specify evenly spaced grid point locations. See also logically rectangular grid, Rectilinear grid, Curvilinear grid.

User component A component that is customized or written by the user. All ESMF components are currently user components. See also generic component.

User time Processor time actually spent executing a PET's code. See also system time, wall clock time.

User transform A user-supplied method that is used to extend framework capabilities beyond generic transforms. See also generic transform.

Virtual Address Space (VAS) A term that refers to the address space in which the computer memory is represented and becomes accessible to an executing PET.

VM Short for Virtual Machine.

Virtual Machine (VM) An ESMF class that abstracts hardware and operating system details. The VM's responsibilities are resource management and topological description of the underlying compute resources in terms of PETs. In addition the VM provides a transparent, low level communication API.

Wall clock time Elapsed real-world time (i.e. difference between start time minus stop time). See also system time, user time.

References

[1] Eaton, B., J. Gregory, B. Drach, K. Taylor, and S. Hankin. NetCDF Climate and Forecast (CF) Metadata Convention. http://cfconventions.org/Data/cf-conventions/cf-conventions-1.6/build/cf-conventions.html, last accessed on Nov 27, 2015.