

# Earth System Modeling Framework

## **ESMF User Guide**

**Version 3.1**

*ESMF Joint Specification Team: V. Balaji, Byron Boville, Samson Cheung, Nancy Collins, Tony Craig, Carlos Cruz, Arlindo da Silva, Cecelia DeLuca, Rosalinda de Fainchtein, Brian Eaton, Bob Hallberg, Tom Henderson, Chris Hill, Mark Iredell, Rob Jacob, Phil Jones, Erik Kluzek, Brian Kauffman, Jay Larson, Peggy Li, Fei Liu, John Michalakes, Sylvia Murphy, David Neckels, Ryan O Kuinghttons, Bob Oehmke, Chuck Panaccione, Jim Rosinski, 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*

25th July 2008

## 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.

- The Spherical Coordinate Remapping and Interpolation Package (SCRIP) from Los Alamos, which informed the design of our regriding functionality
- The Model Coupling Toolkit (MCT) from Argonne National Laboratory, on which we based our sparse matrix multiply approach to general regriding
- 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

<b>1</b>	<b>What is the Earth System Modeling Framework?</b>	<b>4</b>
<b>2</b>	<b>The ESMF User's Guide</b>	<b>4</b>
<b>3</b>	<b>How to Contact User Support and Find Additional Information</b>	<b>5</b>
<b>4</b>	<b>How to Submit Comments, Bug Reports, and Feature Requests</b>	<b>5</b>
<b>5</b>	<b>Quick Start</b>	<b>6</b>
5.1	Downloading ESMF . . . . .	6
5.1.1	From the ESMF web site . . . . .	6
5.1.2	From the SourceForge website . . . . .	6
5.2	Unpacking the download . . . . .	6
5.3	Directory Structure . . . . .	6
5.4	Building ESMF . . . . .	7
5.4.1	Environment Variables . . . . .	7
5.4.2	GNU make . . . . .	8
5.4.3	gmake info . . . . .	8
5.4.4	Building Makefile Targets . . . . .	11
5.4.5	Testing Makefile Targets . . . . .	11
<b>6</b>	<b>Building and Installing the ESMF</b>	<b>12</b>
6.1	ESMF Download Options . . . . .	12
6.2	System Requirements . . . . .	12
6.3	ESMF Environment Variables . . . . .	12
6.4	Supported Platforms . . . . .	17
6.5	Building the ESMF Library . . . . .	19
6.6	Building the ESMF Documentation . . . . .	20
6.7	Installing the ESMF . . . . .	20
<b>7</b>	<b>Porting the ESMF</b>	<b>21</b>
7.1	The ESMF Build System . . . . .	21
7.1.1	General Structure . . . . .	21
7.1.2	Build Configuration . . . . .	22
7.1.3	Source Code Configuration . . . . .	22
7.2	Porting the ESMF to New Platforms . . . . .	23
7.2.1	Customizing the <code>build_rules.mk</code> fragment . . . . .	23
7.2.2	Customizing <code>ESMC_Conf.h</code> and <code>ESMF_Conf.inc</code> . . . . .	27
<b>8</b>	<b>Validating and Running with ESMF</b>	<b>28</b>
8.1	Running ESMF Self-Tests . . . . .	28
8.1.1	Setting up ESMF to run Test Suite Applications . . . . .	28
8.1.2	Running ESMF Unit Tests . . . . .	30
8.1.3	Running ESMF System Tests . . . . .	32
8.2	Running ESMF Examples . . . . .	34
8.2.1	Example Source Code . . . . .	34
8.2.2	Building and Running Examples . . . . .	34
8.3	Using the ESMF . . . . .	36
8.3.1	Shared Object Libraries . . . . .	36
8.3.2	Customized SITE Files . . . . .	36

<b>9 Architectural Overview</b>	<b>38</b>
9.1 Key Concepts	38
9.1.1 Modularity	38
9.1.2 Flexibility	38
9.1.3 Hierarchical Organization	39
9.1.4 Communication Within Components	39
9.1.5 Uniform Communication API	39
9.2 Superstructure	39
9.2.1 Import and Export State Classes	39
9.2.2 Interface Standards	41
9.2.3 Gridded Component Class	41
9.2.4 Coupler Component Class	41
9.2.5 Flexible Data and Control Flow	41
9.3 Infrastructure	43
9.3.1 FieldBundle, Field and Array Classes	43
9.3.2 Grid Class	44
9.3.3 Time and Calendar Management Class	44
9.3.4 Config Resource File Manager	44
9.3.5 DELayout and Virtual Machine	44
9.3.6 Logging and Error Handling	44
9.3.7 I/O Classes	44
<b>10 How to Adapt Applications for ESMF</b>	<b>44</b>
10.1 Individual Components	45
10.2 Full Application	46
<b>11 Glossary</b>	<b>47</b>
<b>References</b>	<b>53</b>

# 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 regriding, 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.

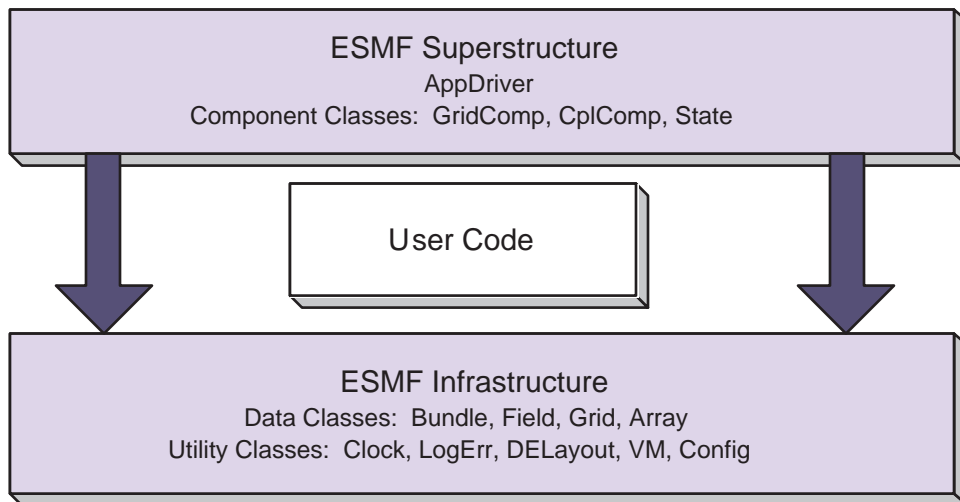
ESMF users may choose to extensively rewrite their codes to take advantage of the ESMF infrastructure, or they may decide to simply wrap their components in the ESMF superstructure in order to utilize framework coupling services. Either way, we encourage users to contact our support team if questions 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 insitutional 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 a quick start program, `quick_start`, that illustrates 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 8 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 operation, such as a description of the directory structure and how to build and run the ESMF examples and quick start programs. Section 9 is an architectural overview that describes the framework’s basic goals and features. Section 10 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.

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.



### 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](mailto:esmf_support@ucar.edu).

More information on the ESMF project as a whole is available on the ESMF website, <http://www.esmf.ucar.edu>. 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](mailto: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. More detailed information on each of these steps, as well as information on running a `quick_start` application and linking the ESMF with your own code is found in Sections 6 and 8.

### 5.1 Downloading ESMF

#### 5.1.1 From the ESMF web site

ESMF is distributed as a source code tar file. The tar file for the latest public release, release notes, known bugs, supported platforms, documentation, and other related information can be found on the ESMF website, under the **Download** tab:

```
http://www.esmf.ucar.edu -> Download
```

The source code for all other releases including the HEAD of the CVS trunk and the last stable version can be found by following the **View All Releases** link on the left hand navigation bar under **Download**:

```
http://www.esmf.ucar.edu -> Download -> View All Releases
```

#### 5.1.2 From the SourceForge website

ESMF can also be downloaded from the SourceForge website from the **Files** link on that website.

```
http://sourceforge.net/projects/esmf -> Files
```

Follow the directions on that web page to download a tar file.

### 5.2 Unpacking the download

The source code comes as a zipped tar file. First unzip the file:

```
gunzip esmf*.tar.gz
```

Then untar the file:

```
tar -xf esmf*.tar
```

This will create a directory called `esmf`.

### 5.3 Directory Structure

The current list of directories includes the following:

- README
- application
- build
- build\_config
- makefile
- scripts
- src

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

## 5.4 Building ESMF

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

1. Set the required environment variables.
2. Type `gmake info` to view and verify your settings
3. Type `gmake` to build the library.
4. Type `gmake check` to run self-tests to verify the build was successful.

See the following sections for more information on each of these steps.

### 5.4.1 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
```

```
cs 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 6 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 and `ESMF_COMM` need not be set. For other systems (e.g. Linux or Darwin) a multitude of MPI implementations is available and `ESMF_COMM` must be set to indicate which implementation is used to build the ESMF library. Set `ESMF_COMM` according to your situation to: `mpich`, `mpich2`, `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.

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 'gmake 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** The SourceForge esmfcontrib repository contains makefiles which have already been customized for certain machines. If one exists for your site and you wish to use it, download the corresponding files into the build\_contrib directory and set ESMF\_SITE to your location (which corresponds to the last part of the directory name). See the SourceForge site <http://sourceforge.net/projects/esmfcontrib> for more information.

**ESMF\_INSTALL\_PREFIX** This variable specifies the prefix of the installation path used during the installation process accessible through 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.

## 5.4.2 GNU make

The ESMF build system uses the GNU make program; it is normally named gmake but may also be simply make or gnumake on some platforms (we will use gmake in this document). ESMF does not use configure or autoconf; the selection of various options is done by setting environment variables before building the framework.

## 5.4.3 gmake info

gmake 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 gmake info:

```
-----  
Make version:  
GNU Make version 3.79, by Richard Stallman and Roland McGrath.  
Built for powerpc-apple-darwin7.0  
Copyright (C) 1988, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99  
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.
```

Report bugs to <bug-make@gnu.org>.

---

Fortran Compiler version:  
ERROR: No input files.  
Pro Fortran 9.0

---

C++ Compiler version:  
g++ (GCC) 3.3 20030304 (Apple Computer, Inc. build 1666)  
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.

---

Preprocessor version:  
gcc (GCC) 3.3 20030304 (Apple Computer, Inc. build 1666)  
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.

---

ESMF\_VERSION\_STRING "3.1.0r beta snapshot"

---

---

\* User set ESMF environment variables \*

ESMF\_OS=Darwin  
ESMF\_DIR=/Users/svasquez/script\_dirs/branch\_builds/esmf  
ESMF\_TESTWITHTHREADS=OFF  
ESMF\_INSTALL\_PREFIX=/Users/svasquez/script\_dirs/branch\_builds/esmf/./install\_dir  
ESMF\_COMM=mpiuni  
ESMF\_TESTEXHAUSTIVE=ON  
ESMF\_SITE=default  
ESMF\_ABI=32  
ESMF\_COMPILER=absoft

---

\* ESMF environment variables \*

ESMF\_DIR: /Users/svasquez/script\_dirs/branch\_builds/esmf  
ESMF\_OS: Darwin  
ESMF\_MACHINE: Power Macintosh  
ESMF\_ABI: 32  
ESMF\_COMPILER: absoft  
ESMF\_BOPT: g  
ESMF\_COMM: mpiuni  
ESMF\_SITE: default

```
ESMF_PTHREADS:      ON
ESMF_ARRAY_LITE:    FALSE
ESMF_NO_INTEGER_1_BYTE: FALSE
ESMF_NO_INTEGER_2_BYTE: FALSE
ESMF_FORTRANSYMBOLS: default
ESMF_TESTEXHAUSTIVE: ON
ESMF_TESTWITHTHREADS: OFF
ESMF_TESTMPMD:      OFF
```

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

-----  
\* ESMF environment variables for final installation \*

```
ESMF_INSTALL_PREFIX:  /Users/svasquez/script_dirs/branch_builds/esmf/./install_dir
ESMF_INSTALL_HEADERDIR: include
ESMF_INSTALL_MODDIR:  mod/modg/Darwin.absoft.32.mpiuni.default
ESMF_INSTALL_LIBDIR:  lib/libg/Darwin.absoft.32.mpiuni.default
ESMF_INSTALL_DOCDIR:  doc
```

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

```
Location of the preprocessor:  /usr/bin/gcc
Location of the Fortran compiler: /Applications/Absoft/bin/f90
Location of the Fortran linker:  /Applications/Absoft/bin/f90
Location of the C++ compiler:    /usr/bin/g++
Location of the C++ linker:      /usr/bin/g++
```

Fortran compiler flags:

```
ESMF_F90COMPILEOPTS: -g -YEXT_NAMES=LCS -YEXT_SFX=_
ESMF_F90COMPILEPATHS: -p/Users/svasquez/script_dirs/branch_builds/esmf/mod/modg/Darwin.absoft
ESMF_F90COMPILECPPFLAGS: -DESMF_TESTEXHAUSTIVE -DS32=1 -DESMF_OS_Darwin=1 -DESMF_MPIUNI
```

Fortran linker flags:

```
ESMF_F90LINKOPTS: -YEXT_NAMES=LCS -YEXT_SFX=_
ESMF_F90LINKPATHS: -L/Users/svasquez/script_dirs/branch_builds/esmf/lib/libg/Darwin.absoft
ESMF_F90LINKRPATHS:
ESMF_F90LINKLIBS: -lstdc++
ESMF_F90ESMFLINKLIBS: -lesmf -lstdc++
```

C++ compiler flags:

```
ESMF_CXXCOMPILEOPTS: -g -DESMF_LOWERCASE_SINGLEUNDERSCORE
ESMF_CXXCOMPILEPATHS: -I/Users/svasquez/script_dirs/branch_builds/esmf/src/include -I/Users
ESMF_CXXCOMPILECPPFLAGS: -DESMF_TESTEXHAUSTIVE -DS32=1 -DESMF_OS_Darwin=1 -D__SDIR__=''
```

C++ linker flags:

```
ESMF_CXXLINKOPTS:
ESMF_CXXLINKPATHS: -L/Users/svasquez/script_dirs/branch_builds/esmf/lib/libg/Darwin.absoft
ESMF_CXXLINKRPATHS:
ESMF_CXXLINKLIBS: -lf90math -lfio -lac -lf77math -lm
```

```
ESMF_CXXESMFLINKLIBS: -lesmf -lf90math -lfio -lac -lf77math -lm
```

```
-----  
Compiling on Mon Apr 7 03:11:21 MDT 2008 on glass.image.ucar.edu  
Machine characteristics: Darwin glass.image.ucar.edu 7.9.0 Darwin Kernel Version 7.9.0: Wed  
-----
```

#### 5.4.4 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 6 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

#### 5.4.5 Testing Makefile Targets

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

```
gmake check
```

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

See section 8.1.1 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

**build\_all\_tests** build tests only, do not execute

**run\_all\_tests** run tests without rebuilding and print a summary of the results

**check\_all\_tests** print out the results summary without re-executing the tests again

**clean\_all\_tests** remove all test executables

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, `gmake build_examples` recompiles the example programs but does not execute them. `gmake 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.

## 6 Building and Installing the ESMF

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

### 6.1 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.esmf.ucar.edu>.

The ESMF is distributed as a full source code tree. You will need to compile the code into the `libesmf.a` library. On some platforms a shared library, `libesmf.so`, is also created. Follow the instructions in the following sections to build the library and link it with your application.

### 6.2 System Requirements

The following compilers and utilities are required for compiling, linking and testing the ESMF software:

- Fortran90 (or later) compiler;
- C++ compiler;
- MPI implementation compatible with the above compilers (but see below);
- GNU's gcc compiler - for a standard cpp preprocessor implementation;
- GNU make;
- Perl - for running test scripts.

Alternatively ESMF can be built using a single-processor MPI-bypass library that comes with ESMF. It allows ESMF applications to be linked and run in single-process mode.

In order to build html and pdf version of the ESMF documentation,  $\LaTeX$ , the `latex2html` conversion utility, and the Unix/Linux `dvipdf` utility must be installed.

### 6.3 ESMF Environment Variables

The following is a full alphabetical list of all environment variables which are used by the ESMF build system. In many cases only `ESMF_DIR` must be set. On Linux and Darwin systems `ESMF_COMPILER` and `ESMF_COMM` must also be set to select the appropriate Fortran and C++ compilers and MPI implementation. The other variables have default values which work for most systems.

**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. If no data of rank greater than 4D will be used, setting this variable to any value will prevent ESMF from generating interfaces for 5D to 7D arrays. This will shrink the amount of autogenerated code.

**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\_COMM** Possible value: *system-dependent*

On systems with a vendor-supplied MPI communications library the vendor library is chosen by default for communications and ESMF\_COMM need not be set. For other systems (e.g. Linux or Darwin) a multitude of MPI implementations is available and ESMF\_COMM must be set to indicate which implementation is used to build the ESMF library. Set ESMF\_COMM according to your situation to: mpich, mpich2, 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.

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) 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 'gmake 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: *flag*

This variable can be used to prepend flags to default compiler flags.

**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\_CXXLINKDIRS** Possible value: *flag*

This variable can be used to prepend directories to default linker directories.

**ESMF\_CXXLINKLIBS** Possible value: *flag*

This variable can be used to prepend libraries to default linker libraries.

**ESMF\_CXXLINKOPTS** Possible value: *flag*

This variable can be used to prepend flags to default linker flags.

**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\_CXXOPTFLAG** Possible value: *flag*

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

**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: *flag*

This variable can be used to prepend flags to default compiler flags.

**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 used to specify the F90 module directory.

**ESMF\_F90LINKDIRS** Possible value: *flag*

This variable can be used to prepend directories to default linker directories.

**ESMF\_F90LINKLIBS** Possible value: *flag*

This variable can be used to prepend libraries to default linker libraries.

**ESMF\_F90LINKOPTS** Possible value: *flag*

This variable can be used to prepend flags to default linker flags.

**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\_F90OPTFLAG** Possible value: *flag*

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

**ESMF\_INSTALL\_PREFIX** Possible value: *relative or absolute path*

This variable specifies the prefix of the installation path used during the installation process accessible through 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.

**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 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 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 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 or relative to ESMF\_INSTALL\_PREFIX.

**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 8.1.1 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 8.1.1 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 8.1.1 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 8.1.1 for a discussion of this option.

**ESMF\_NETCDF\_INCLUDE** Possible value: *absolute path*

This variable can be used to point to the desired NETCDF header files.

**ESMF\_NETCDF\_LIBPATH** Possible value: *absolute path*

This variable can be used to point to the desired NETCDF library files.

**ESMF\_NO\_INTEGER\_1\_BYTE** Possible value: TRUE, FALSE (default)

Not normally set by user. Setting this variable to ON will prevent ESMF from generating data array interfaces for data types of 1-byte integers.

**ESMF\_NO\_INTEGER\_2\_BYTE** Possible value: TRUE, FALSE (default)

Same as `ESMF_NO_INTEGER_1_BYTE` but for 2-byte integers.

**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\_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.)

The Sourceforge `esmfc contrib` repository contains makefiles which have already been customized for certain machines. If one exists for your site and you wish to use it, download the corresponding files into the

build\_contrib directory and set ESMF\_SITE to your location (which corresponds to the last part of the directory name). See the Sourceforge site <http://sourceforge.net/projects/esmfcontrib> for more information.

**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\_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\_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.

Environment variables must be set in the user's shell or when calling gmake. 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 gmake command line:

```
gmake ESMF_ABI=32
```

## 6.4 Supported Platforms

The following two tables list various combinations of environment variable settings used by the ESMF build system. A default value in the compiler column indicates the vendor compiler. A mpi value in the comm column indicates the vendor MPI implementation.

The first table lists the exact combinations which are tested regularly and are fully supported. The second table lists all possible combinations which are included in the build system.

**Fully tested combinations:** (See <http://www.esmf.ucar.edu/download/platforms/> for the most up-to-date table of supported combinations.)

	<b>ESMF_OS</b>	<b>ESMF_COMPILER</b>	<i>F90 compiler</i>	<i>C++ compiler</i>	<b>ESMF_COMM</b>	<b>ESMF_ABI</b>
Cray X1E	Unicos	default	ftn 5.5.0.9	CC 5.5.0.9	mpi	64
Cray XT4	Unicos	pgi	ftn 7.1-6	CC 7.1-6	mpi	64
HP ZX6000	Linux	intel	ifort 8.1.021	icpc 8.1.024	lam	64
IBM Bluegene	Linux	xlf	mpxlf90 10.1.0.4	mpxlc 8.0.0.4	mpi	32
IBM Opteron	Linux	pathscale	pathf90 2.4	pathCC 2.4	mpich	x86_64_small, x86_64_medium
IBM Opteron	Linux	pgi	pgf90 6.2-3	pgCC 6.2-3	mpich	x86_64_small, x86_64_medium
IBM SP	AIX	default	mpxlf90_r 11.1.0.0	mpCC_r 9.0.0.1	mpi	32, 64
IBM SP	AIX	default	mpxlf90_r 11.1.0.0	mpCC_r 9.0.0.2	mpi	32, 64
Mac G5	Darwin	absoft	f90 9.0	g++ 3.3	lam, mpiuni	32
Mac G5	Darwin	absoft	f90 8.2	g++ 3.3	lam, mpiuni	32
Mac G5	Darwin	nag	f95 5.0(272)	g++ 3.3	lam, mpiuni	32
Mac G5	Darwin	xlf	xlf90_r 8.1	xlC_r 6.0	lam, mpiuni	32
Mac G5	Darwin	xlfgcc	xlf90_r 8.1	g++ 3.3	lam, mpiuni	32
Mac x86_64	Darwin	intel	ifort 10.1.008	icpc 10.1.009	openmpi	64
PC Pentium III	Linux	lahey	lf95 L6.20d	g++ 4.1.2	mpiuni	32
PC Pentium III	Linux	pgi	pgf90 7.1-3	pgCC 7.1-3	mpiuni	32
PC Pentium III	Linux	pgigcc	pgf90 7.1-3	g++ 4.1.2	mpiuni	32
PC Pentium M	Cygwin	gfortran	gfortran 4.4.0	g++ 3.4.4	mpiuni	32
PC Xeon (32) Cluster	Linux	absoft	f90 9.0 r2	g++ 3.2.3	mpich	32
PC Xeon (32) Cluster	Linux	g95	g95 4.0.2	g++ 4.0.2	mpich	32
PC Xeon (32) Cluster	Linux	intel	ifort 8.1	icpc 8.1	mpich	32
PC Xeon (32) Cluster	Linux	nagintel	f95 5.0(361)	icpc 8.1	mpich	32
PC Xeon (64)	Linux	gfortran	gfortran 4.3.1	g++ 3.3.3	mpiuni	64, x86_64_medium
PC Xeon (64) Cluster	Linux	intel	ifort 9.1.036	icpc 9.1.042	mpich2	64
PC Xeon (64) Cluster	Linux	pgi	pgf90 7.1-3	pgCC 7.1-3	mpich2	64
PC Xeon (64) Cluster	Linux	intel	ifort 10.0.025	icpc 10.0.025	scalimpi	64
PC Xeon (64) Cluster	Linux	pgi	pgf90 6.2-4	pgCC 6.2-4	scalimpi	64
PC Xeon (64) Cluster	Linux	intel	ifort 9.1.032	icpc 9.1.038	intelmpi	64
SGI Altix	Linux	intel	ifort 9.1.045	icpc 9.1.049	mpi	64
SGI Origin 3800	IRIX64	default	f90 7.4.4m	CC 7.4.4m	mpi	64
Sun SPARC	SunOS	default	f95 8.3	CC 5.9	mpi	64

**All possible options.** Where multiple options exist and the default is independent of ESMF\_MACHINE the default value is in **bold**:

ESMF_OS	ESMF_COMPILER	ESMF_COMM	ESMF_ABI
AIX	default	mpiuni, <b>mpi</b> , user	32, <b>64</b>
Cygwin	g95	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user	32, 64
Cygwin	gfortran	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user	32, 64
Darwin	absoft	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user	32
Darwin	g95	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user	<b>32</b> , 64
Darwin	gfortran	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user	<b>32</b> , 64
Darwin	intel	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user	<b>32</b> , 64
Darwin	intelgcc	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user	<b>32</b> , 64
Darwin	nag	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user	32
Darwin	xlf	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user	32
Darwin	xlfgcc	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user	32
IRIX64	default	mpiuni, <b>mpi</b> , user	32, <b>64</b>
Linux	absoft	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user	32, 64
Linux	absoftintel	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user	32, 64
Linux	g95	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user	32, 64, ia64_64, x86_64_32, x86_64_small, x86_64_medium
Linux	gfortran	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user	32, 64, ia64_64, x86_64_32, x86_64_small, x86_64_medium
Linux	intel	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user, intelmpi, scalimpi	32, 64, ia64_64, x86_64_32, x86_64_small, x86_64_medium
Linux	intelgcc	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user, intelmpi	32, 64, ia64_64, x86_64_32, x86_64_small, x86_64_medium
Linux	lahey	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user	32
Linux	nag	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user	32
Linux	nagintel	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user	32
Linux	pathscale	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user	32, 64, x86_64_32, x86_64_small, x86_64_medium
Linux	pgi	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user scalimpi	32, 64, x86_64_32, x86_64_small, x86_64_medium
Linux	pgigcc	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user	32
Linux	xlf	<b>mpiuni</b> , mpich, mpich2, lam, openmpi, user	32
OSF1	default	mpiuni, <b>mpi</b> , user	64
SunOS	default	mpiuni, <b>mpi</b> , user	32, <b>64</b>
Unicos	default	mpiuni, <b>mpi</b> , user	64
Unicos	pgi	mpiuni, <b>mpi</b> , 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.

## 6.5 Building the ESMF Library

GNU make is required to build the library. On some systems this will be just the command make. On others it might be installed as gmake or even gnumake. In any event, use the `-version` option with the make command to determine if it is GNU make.

Build the library with the command:

```
gmake
```

Makefiles throughout the framework are configured to allow users to compile files only in the directory where `gmake` is entered. Shared libraries are rebuilt only if necessary. In addition the entire ESMF framework may be built from any directory by entering `gmake all`, assuming that all the environmental variables are set correctly as described in Section 6.3.

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 `gmake run_examples` or `gmake run_unit_tests`, respectively. For non-multiprocessor machines, uni-processor targets are available as `gmake run_examples_uni` or `gmake run_unit_tests_uni`.

## 6.6 Building the ESMF Documentation

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

The tarballs on the ESMF website for ESMF versions 3.0.1 and later do not contain the ESMF documentation files. The documentation is available on the ESMF website in html or pdf form and most users should not need to build it from the source.

If a user does want to build the documentation, they will need to download the `esmfc` module from the ESMF SourceForge repository (see section 5.1.1. `Latex` and `latex2html` must be installed.

To build documentation:

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

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

## 6.7 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 five `ESMF_` environment variables:

- `ESMF_INSTALL_PREFIX`
- `ESMF_INSTALL_HEADERDIR`
- `ESMF_INSTALL_LIBDIR`
- `ESMF_INSTALL_MODDIR`
- `ESMF_INSTALL_DOCDIR`

Install ESMF with the command:

```
gmake install
```

Check the ESMF installation with the command:

```
gmake installcheck
```

## 7 Porting the ESMF

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

### 7.1 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.

#### 7.1.1 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 6.3 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.

### 7.1.2 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
```

### 7.1.3 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 by 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.

## 7.2 Porting the 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.

### 7.2.1 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\_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\_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\_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\_UTCSRIPTS**  
**ESMF\_WC**

### **7.2.2 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(func)** Macro that will correctly expand "func" to match the Fortran symbol convention.

**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++.

**ESMF\_IS\_32BIT\_MACHINE** This macro will be defined on a 32-bit architecture.

**ESMF\_IS\_64BIT\_MACHINE** This macro will be defined on a 64-bit architecture.

**ESMF\_F90\_PTR\_BASE\_SIZE** Value in bytes used to calculate the size of the Fortran dope vector.

**ESMF\_F90\_PTR\_PLUS\_RANK** Value in bytes used to calculate the size of the Fortran dope vector.

**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.

## 8 Validating and Running with ESMF

This section goes into more detail about how to run the tests which are included with the ESMF software, how to run the examples and `quick_start` software, and how to link your own applications with ESMF.

### 8.1 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.

Test targets will compile the ESMF library if it has not already been built.

#### 8.1.1 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`.

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.

Finally, 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 parallel a specific application needs to be run.

Two modes of operation need to be considered for the ESMF batch access. First, if interactive batch access is available, it is straight forward 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 through 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.esmf.ucar.edu/download/platforms/> web page and follow the link for the platform of interest. Each test report contains the output of `gmake info`, which lists the settings of the `ESMF_MPIxxx` environment variables.

Furthermore, the `esmfcontrib` repository on SourceForge hosts a `scripts` module that contains scripts that were customized to certain user environments. These scripts are not generic enough to be included in the ESMF distribution, but users faced with the need to customize a script to their environment may find the script collection on `esmfcontrib` a valuable resource. Please refer to <https://sourceforge.net/projects/esmfcontrib> on how to access `esmfcontrib`, and the `scripts` module in particular.

### 8.1.2 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:

```
gmake [ ESMF_TESTEXHAUSTIVE=<ON,OFF> ] tests
gmake [ ESMF_TESTEXHAUSTIVE=<ON,OFF> ] 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
gmake 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. The following is a sample from the script output:

There are a total of 1224 exhaustive multi processor unit tests, 1220 pass and 4 fail.

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/IOSpec/tests/ESMF_IOSpecUTest.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
```

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
```

### 8.1.3 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.esmf.ucar.edu>. 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:

```
gmake [SYSTEM_TEST=xxx] system_tests
gmake [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>
gmake 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/test0/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. The following is a sample from the script output:

```
There are 14 system tests, 12 passed and 2 failed.
```

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

## 8.2 Running ESMF Examples

### 8.2.1 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.

### 8.2.2 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:

```
gmake 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 examples failures. The following is a sample from the script output:

There are 34 examples, 32 passed and 2 failed.

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/examples0/AIX.default.64.de
```

## 8.3 Using the ESMF

To use an ESMF library to compile and link your application against you will need to tell the compiler where the ESMF module and ESMF library files are located. If this has already been documented by the installer of the ESMF library, follow the directions given. However, if not, then you must add the correct compiler and linker flags yourself.

In order to compile a Fortran application against ESMF add the directory that contains the ESMF \*.mod file(s):

```
$ESMF_DIR/mod/mod$ESMF_BOPT/$ESMF_OS.$ESMF_COMPILER.$ESMF_ABI.$ESMF_SITE
```

For most Fortran compilers this path is identified either with a `-I` or a `-M`. Linking against the ESMF library is a bit more complex because of ESMF's dependency on the Fortran90 *and* C++ runtime libraries. Generally it is not enough to specify the location of the ESMF library:

```
$ESMF_DIR/lib/lib$ESMF_BOPT/$ESMF_OS.$ESMF_COMPILER.$ESMF_ABI.$ESMF_SITE
```

via the `-L` flag followed by `-lesmf!` However, ESMF provides a file called `esmf.mk` that is generated during the library build and stored in the same location as the ESMF library (see path just above). This makefile fragment defines several variables indicating compiler options, include paths, library paths and libraries necessary to compile and link against ESMF with Fortran or C++ compilers. Look at "esmf.mk" to pull out the necessary flags to compile and link your own code against the installed ESMF library.

Alternatively, you may want to include "esmf.mk" from within your own build system. All of the variables defined in "esmf.mk" have prefix "ESMF\_" as to prevent name space conflicts with users' makefiles. Notice that "esmf.mk" is a self-contained file and is not affected by environment variables. It is not necessary to set any ESMF\_ environment variables to use an ESMF library that has been installed on your system!

When building the ESMF libraries on platforms that support both 32 and 64 bit application binary interfaces, verify that the ESMF\_ABI environment variable is set to match the compile option that was specified to build your application.

### 8.3.1 Shared Object Libraries

On some platforms, a shared object library is created in addition to the standard .a archive library. Shared object libraries are libraries that are loaded by the first program that uses them. All programs that start afterwards automatically use the existing shared library. The library is kept in memory as long as any active program is still using 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.

### 8.3.2 Customized SITE Files

In an effort to provide platform specific information for building ESMF and linking the libraries with your application, a SourceForge site, `esmfcontrib`, has been created. To locate the platform makefiles for a specific institution, check out the `build_config_files` using the appropriate CVSROOT. The URL for the `esmfcontrib` SourceForge site is:

`http://sourceforge.net/projects/esmfcontrib/`

Additionally, you may check out all the platform makefile fragments for a particular institution from the `esmfcontrib` site. For example, to check out the available makefile fragments for platforms at the National Center for Atmospheric Research, `ncar`, change directories to

```
$ESMF_DIR/build_config
```

and use the following CVS command:

```
cvs -z3 -d:ext:$username@cvs.sourceforge.net:/cvsroot/esmfcontrib checkout ncar
```

The following directories will be checked out:

```
AIX.default.bluesky  
Linux.lahey.longs
```

To build using these makefiles you must set the environment variable `ESMF_SITE` to `bluesky`, or `longs`.

At the present time, we have files for the following institutions:

```
anl - Argonne National Laboratory  
cola - Center for Ocean-Land-Atmosphere Studies  
gsfc - Goddard Space Flight Center  
mit - Massachusetts Institute of Technology  
ncar - National Center for Atmospheric Research
```

Users are encouraged to contribute pertinent information to the `esmfcontrib` repository.

## 9 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 9.2. 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 9.3. A hierarchical combination of superstructure, user code components, and infrastructure are joined together to form an ESMF application.

### 9.1 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.

#### 9.1.1 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 its 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 (not yet implemented) are called Transforms, and they are 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.**

#### 9.1.2 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.**

### 9.1.3 Hierarchical Organization

The 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 doesn't 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's 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.**

### 9.1.4 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.**

### 9.1.5 Uniform Communication API

ESMF is developing 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's 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.**

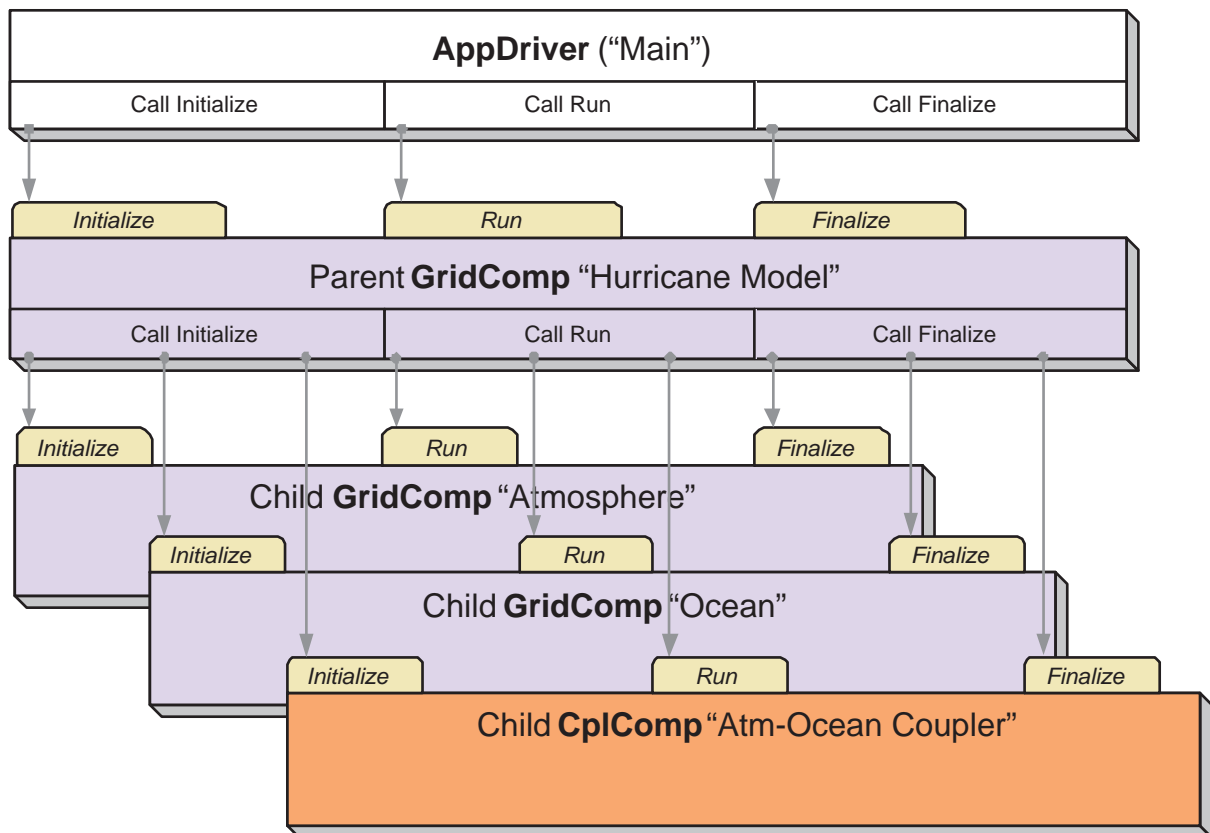
## 9.2 Superstructure

The ESMF superstructure layers in an application furnish 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. We describe these classes below:

### 9.2.1 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

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 9.1.3.



code components to, for example, fill an export State object with data to be shared with other components or query an 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 very large data structures can be avoided where possible. The content of an import State and an export State can be made self-describing.

### **9.2.2 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; instead it provides 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.

### **9.2.3 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.

### **9.2.4 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.

### **9.2.5 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.

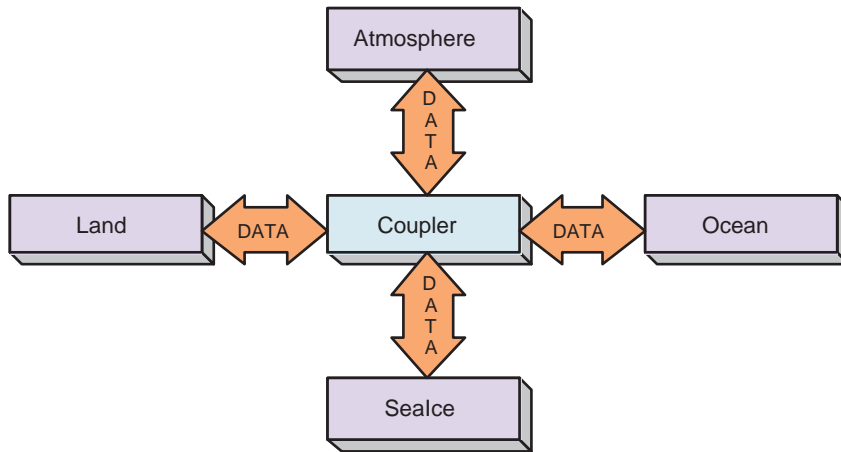


Figure 4: ESMF also supports configurations with multiple point to point Coupler Components. These take inputs from one Gridded Component and transfer and regrid the data before passing it to another Gridded Component. This schematic shows a flow of data between two Coupler Components that connect three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

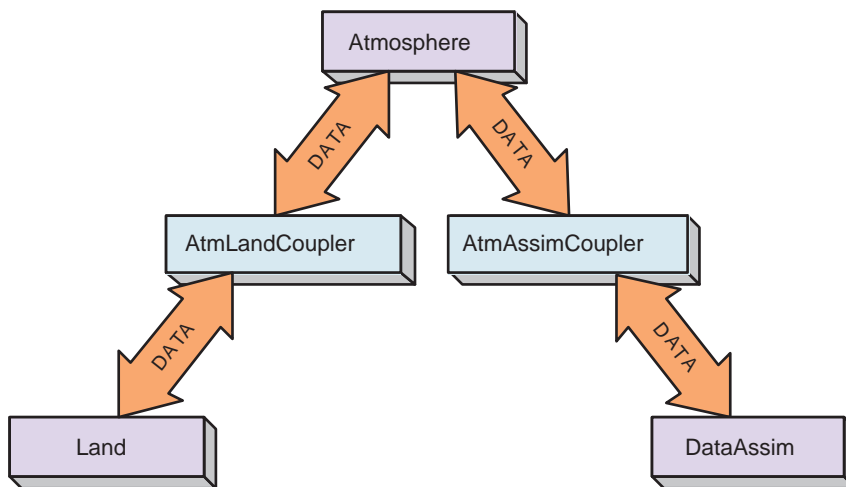
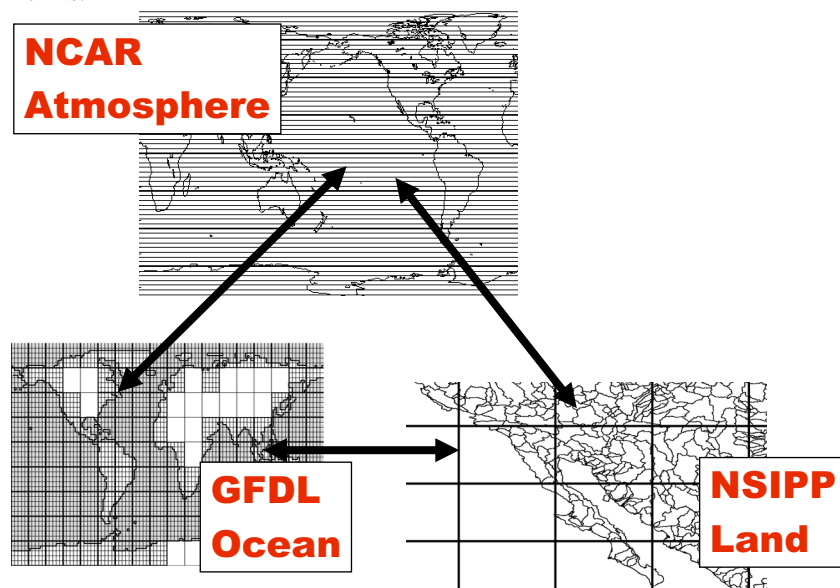


Figure 5: Schematic showing the coupling of components that use different discrete grids and different time-stepping. In this example, component *NCAR Atmosphere* might use a spectral grid based on spherical harmonics, component *GFDL Ocean* might use a latitude-longitude grid but with a patched decomposition that does not include land masses and component *NSIPP Land* might use a mosaic-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.



### 9.3 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. We describe these classes below:

#### 9.3.1 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, not yet implemented, will provide a self-describing container for unstructured observational data streams.

### 9.3.2 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.

### 9.3.3 Time and Calendar Management Class

To support synchronization between components *Time*, *Time Interval*, *Calendar*, *Clock*, and *Alarm* classes are provided. 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.

### 9.3.4 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.

### 9.3.5 DELayout and Virtual Machine

To provide a mechanism for ensuring performance portability ESMF defines *DELAYOUT* and *Virtual Machine* classes. These classes provide a set of high-level 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.

### 9.3.6 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.

### 9.3.7 I/O Classes

The infrastructure layer will define a set of *I/O* classes for storing and retrieving *Field* and *Grid* information to and from persistent storage. *I/O* classes are not yet implemented.

## 10 How to Adapt Applications for ESMF

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

## 10.1 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.

## 10.2 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 an unchanged copy of the file found in the `AppDriver` directory. The only customization needed is to set the name of the top level Gridded Component, and to set the name of the `SetServices` routine. The template file includes a call to `ESMF_Initialize()` which ensures the framework initialization code is run, and will provide the environment for components to be created and run.

Although ESMF provides 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.

## 11 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:

<u>date</u>	<u>day of year</u>
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 and GRIB2 home pages.

**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 Physical Grid, 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.

**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 glos:ConcurrentExecution.

**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://www.cgd.ucar.edu/cms/eaton/cf-metadata/index.html>.