Jump-starting the development of coupled climate models with minimal effort using a new communication library

Raffaele Montuoro
Department of Atmospheric Sciences
Texas A&M University
Outline

- Why a new library? Overview of available tools for building coupled models and practical challenges
- The Texas A&M Coupling Library (AMC): Foundations & Introduction to the API
- Hands-on exercise: let’s build a coupled model!
Overview

Software tools available for building coupled models today
Available coupling software and tools

Major coupling frameworks:

**CPL7** — NCAR model coupler, version 7 (Craig et al., 2012)
Included in the Community Climate System Model (CCSM4) and in the Community Earth System Model (CESM1) based on the Model Coupling Toolkit

**OASIS** — CERFACS/CNRS (France) coupler (Valcke et al., 2006)
Originally based on the Prism System Model Interface (PSMIle)

**OASIS3-MCT** released on May 28, 2013

**FMS** — NOAA GFDL Flexible Modeling System (Balaji, 2004)
Communication kernels: *MPP modules*, built on MPI/SHMEM/NUMA

**ESMF** — Earth System Model Framework (started in 2002)
Based on CCSM, FMS, , and more...
Available coupling software and tools

**FOAM** — custom coupler in Fast Ocean Atmosphere Model
ANL-UW, started 1994; development frozen in 2002 (version 1.5)

**OpenPALM** — robust coupler for multi-physics models (2011)
Supports industrial codes via TCP/IP connections

**FLUME** — FLexible Unified Model Environment (Ford & Riley, 2002)
Built specifically for the UK Met Office Unified Model System

—Is a new coupling framework necessary?
Why a new coupling library?

— Are we going to reinvent the wheel?

...AND I HAVE FOUND THIS ONE WORKS A LOT BETTER.

http://tradingeducationblogs.com/day-trading-mistakes/
Motivation

- Available coupling tools are highly complex—their manipulation requires expert software engineers.
- Advances in science often require testing unconventional hypotheses.
- Academic research usually doesn’t involve teams of expert software engineers.
- Is it possible to create a model-coupling tool of minimal complexity that can be quickly learned by researchers with diverse backgrounds and interests?
The Texas A&M Coupling Library (AMC)

Foundations and API
AMC: Foundations

AMC is a parallel coupling library conceived to enable data exchange between individual programs (components) with minimal coding.

- AMC’s design is based on a distributed-memory model: each parallel task is assumed to have access only to its individual memory space.

- Given its design, AMC’s implementation using the Message Passing Interface (MPI) is straightforward.

- AMC is written in standard Fortran 90.

- Disclaimer: AMC is work in progress.
AMC: Foundations

- AMC’s architecture relies on:
  - A communication framework (driver/dispatcher)
  - Coupled components

- All communications between components are routed through a **hub** (framework)

- The hub is *solely* responsible for handling all communications (dispatch)
AMC: Foundations

In MPI language:

Framework
global communicator

- Components do not overlap
- The order of components and components tasks can be chosen
- Identity of root task can be assigned in each component
AMC: Foundations

- Each AMC parallel task has a dual identity:
  - It belongs to the framework
  - It belongs to the component

- Task identities are fully handled internally
  e.g.: local vs. remote memory access

- Public variables are provided to identify tasks:
  - Framework IDs may be accessed by all components
  - Component IDs are defined only inside the component
The AMC library provides the following variables to identify each parallel task:

**Framework**

integer :: &
  amc_comm, & ! global comm
  amc_root, & ! root task
  amc_rank, & ! rank id
  amc_size, & ! total # tasks
  amc_io_rank ! id of I/O task

logical :: &
  amc_is_io, & ! is the I/O task?
  amc_is_root ! is root task?

**Component**

integer :: &
  amc_cmp_comm, & ! Local comm
  amc_cmp_root, & ! root task
  amc_cmp_rank, & ! rank id
  amc_cmp_size, & ! # cmp tasks
  amc_cmp_io_rank ! id of I/O task

logical :: &
  amc_cmp_is_io, & ! is I/O task?
  amc_cmp_is_root ! is root task?
AMC: Syntax

Basic syntax rules:

- `amc_` symbols are used in framework
- `amc_cmp_` symbols are used in components

Names of functions and subroutines follow the rule:

Framework:
- `amc_<object>_method`

Component:
- `amc_cmp_<object>_method`
A **minimal** set of calls is required to connect/disconnect a model component to/from the framework.

**Reminder:** All connections and communications are handled by the framework (dispatcher).

```fortran
use amc

integer :: n1  ! N. tasks comp. 1
integer :: n2  ! N. tasks comp. 2
integer :: rc   ! Return code

call amc_init(rc)

call amc_frame_setup((/n1,n2/),rc)

call amc_frame_connect(req,rc)  ! connections are asynchronous

call amc_req_complete(req)
```

```fortran
use amc

integer :: root   ! Rank of root
integer :: comm   ! communicator
integer :: rc     ! Return code

comm = amc_cmp_comm

! Init component and connect to framework (blocking)
call amc_cmp_init(root,comm,rc)
```
Communications (data exchanges) are carried out as:

_infol_ unstructured communications
informational data exchanges, not necessarily related to each other

_stream_ structured communications
data exchanges follow known patterns

**Example:** surface fluxes between ocean and atmosphere
AMC: Routing

Communications are **always** routed through the framework (dispatcher):

- **info**
  - unstructured communications may occur only between a single framework task \(\textit{root}\) and a single component task \(\textit{root}\).

- **stream**
  - structured communications occur between a single framework task \(\textit{root}\) and all the tasks of a given component via the component’s \(\textit{root}\) task.
To summarize:

**_info_** unstructured communications

AMC

one-to-one

one-to-one

cmp

**_stream_** structured communications

AMC

one-to-many

many-to-one

cmp
AMC: Streams

- **Multiple** streams can be opened for each component.
- Each stream can be configured with its unique data packing/unpacking method.
- Each stream includes a regridding procedure based on the SCRIP package (Jones, 1998). The regriddler can be configured, then turned on or off if needed.
- Multiple streams per component may be used to couple multiple domains in nested models.
AMC: Streams

- Streams need to be created in each component
- Streams must be connected to the frameworks to allow communication with the component
- Stream connections are handled exclusively by the framework
- All streams in a component are connected at once
- Each component’s stream is identified by a unique integer id assigned at creation time
AMC: Streams

Example:

```fortran
integer :: id, idx(:,), n, rc

n = 0
do i = ibeg, iend  ! tile bounds (tiled domain)
    n = n + 1
    idx(n) = i  ! build address array
end do

id = 2  ! set the new stream id to 2
call amc_cmp_stream_create(id, idx, n, rc)
```

```fortran
integer :: cmp_id, rc, req(:) ! N. tasks comp. 1

cmp_id = 1  ! connect all streams from component 1

call amc_stream_connect(cmp_id, req, rc)
! connections are asynchronous

call amc_req_complete(req)
```
AMC: Communicate

- Generic send/receive methods provided:
  - `_get()` retrieve method (remote to local)
  - `_set()` send method (local to remote)

- Call syntax may be “fully transparent”
  - Symbols names are identical on both ends when buffers declaration statements are included in a common module
  - Buffers at end side are automatically allocated, if needed

- `_info_` communications are asynchronous
  They can be aggregated to improve performance
AMC: Receive info

shared module

module buffers
    integer :: rbuflen ! Length of message buffer
    real, dimension(:), pointer :: rbuf ! frame<->cmp
end module buffers

use buffers
allocate rbuf(rbuflen)

rbuf(:) = localdata(:) ! pack data into buffer

component

use buffers
integer :: cmp_id, rc, req(:)

cmp_id = 1 ! receive data from component 1
call amc_info_get(rbuflen, cmp_id, req, rc)
call amc_req_complete(req) ! connections are asynchronous

Framework

call amc_info_get(rbuf, rbuflen, cmp_id, req, rc)
call amc_req_complete(req)
module buffers
    integer :: sbuflen  ! Length of message buffer
    real, dimension(:), pointer :: sbuf  ! frame->cmp
end module buffers

use buffers
integer :: cmp_id, rc, req(:)
allocate sbuf(sbuflen)
sbuf(:) = ...

cmp_id = 2  ! send data to component 2
call amc_info_set(sbuf, buflen, cmp_id, req, rc)
call amc_req_complete(req)  ! connections are asynchronous

use buffers
localdata(:) = sbuf(:)  ! Use received data
**AMC: Receive data stream**

**Module_buffers**
```
module buffers
  ! global & local receive buffers: cmp->frame
  real, dimension(:,), pointer :: rbuf_glob, rbuf_loc
end module buffers
```

**Component**
```
use buffers
integer :: i, n

n = 0
do i = ibeg, iend  ! tile bounds (tiled domain)
  n = n + 1
  rbuf_loc(n) = localdata(i)
end do
```

**Framework**
```
use buffers
integer :: id, cmp_id, rc

cmp_id = 1  ! receive data from component 1
id = 2      ! receive data from stream 2 of component 1

call amc_stream_get(id, rbuf_glob, rbuf_loc, cmp_id, rc)
```
AMC: Send data stream

module buffers
  ! global & local receive buffers
  real, dimension(:), pointer :: sbuf_glob, sbuf_loc
end module buffers

use buffers
integer :: id, cmp_id, rc

cmp_id = 3  ! send data to component 3
id = 1      ! send data to stream 1 of component 3
sbuf_glob(:) = globaldata(:)

call amc_stream_set(id, sbuf_glob, sbuf_loc, cmp_id, rc)

use buffers
allocate rbuf_loc(iend-ibeg+1)  ! allocate buffer on local tile
rbuf_loc(:) = localdata(:)     ! load tiled data chunk
NOTE: Since all communications are initiated by the framework (dispatcher), **strict synchronization is required** between components and framework.

```
call amc_sync(rc)
! exchange data (get/set)
call amc_sync(rc)
```
AMC: Regridding capabilities

- A regridding procedure based on SCRIP (Jones, 1998) is embedded in each _stream_

- It can be referenced using the object name: _stream_map_

- Regridding of a data stream can be setup and activated/deactivated using the following methods:
  - _load()_ Loads regridding parameters (weights, grid data)
  - _switch()_ Turns regridding on/off

**NOTE:** _stream_map_ can only be used by root task in framework
AMC: Regridding capability

Regrid data from framework to component

! load regridding data (SCRIP) to stream id in ! component cmp_id for frame->cmp regridding
call amc_stream_map_load(id, wts, num_wts, num_lnk, &
  dst_add, dst_grd_size, &
  src_add, src_grd_size, to = cmp_id)

! activate regridding
call amc_stream_map_switch(id, cmp_id, .true.)

Regrid data from component to framework

! load regridding data (SCRIP) to stream id in ! component cmp_id for frame->cmp regridding
call amc_stream_map_load(id, wts, num_wts, num_lnk, &
  dst_add, dst_grd_size, &
  src_add, src_grd_size, from = cmp_id)

! activate regridding
call amc_stream_map_switch(id, cmp_id, .true.)
Hands-on exercise:
Let’s build a coupled model!

How to build a basic coupled model using AMC
Building a basic coupled model

Three parts:

1. Model driver
2. Component 1 (e.g. atmosphere)
3. Component 2 (e.g. ocean)
Building a basic coupled model

Model driver

```fortran
program drv
  integer :: rc

  call drv_init(rc)
  if (rc.eq.0) call drv_run(rc)
  call drv_finalize(rc)

end program drv
```

```fortran
subroutine drv_init(rc)
  use amc
  use buffers
  integer :: n1, n2, rc, req(:)

  call amc_init(rc)
  call amc_frame_setup((/n1,n2/), rc)
  call amc_frame_connect(req,rc)
  select case (amc_cmp_id)
    case (1)
      call atm_init(rc)
    case (2)
      call ocn_init(rc)
  end select
  call amc_req_complete(req)
  call amc_stream_connect(1, req, rc)
  call amc_stream_connect(2, req, rc)
  call amc_req_complete(req)

end subroutine drv_init
```

```fortran
subroutine drv_finalize
  use amc

  call amc_finalize

end subroutine drv_finalize
```
Building a basic coupled model

```fortran
subroutine drv_run(rc)
  use amc
  use buffers

  do ! main time loop
    select case (amc_cmp_id)
      case (1)
        call atm_import(rc)
        call atm_run(rc)
        call atm_export(rc)
      case (2)
        call ocn_import(rc)
        call ocn_run(rc)
        call ocn_export(rc)
    end select
    call amc_sync(rc)
    call amc_stream_get(1, atm_buf_g, atm_buf, 1, rc)
    call amc_stream_set(1, ocn_buf_g, ocn_buf, 2, rc)
    call amc_sync(rc)
  end do
end subroutine drv_run
```
Building a basic coupled model

- Layout of a model component (atmosphere)

```fortran
subroutine atm_init(rc)
  use amc
  use buffers
  integer :: comm, root, rc, req(:)

  comm = amc_cmp_comm
  call amc_cmp_init(root, comm, rc)

  n = 0
  do i = ibeg, iend
    n = n + 1
    idx(n) = i
  end do

  call amc_cmp_stream_create(1, idx, n, rc)
end subroutine atm_init
```

```fortran
subroutine atm_run(rc)
  ! run model
end subroutine atm_run
```

```fortran
subroutine atm_import(rc)
  use buffers
  localdata(:) = recvbuf(:)
end subroutine atm_import
```

```fortran
subroutine atm_export(rc)
  use buffers
  sendbuf(:) = localdata(:)
end subroutine atm_export
```
Future work

- Finalize first public release of AMC
- Parallelize regridding in streams
- Implement collective communications
- Build a state-of-the-art coupled regional climate model for research
Questions?

Thank you!

rmontuoro@tamu.edu