MPAS
Model for Prediction Across Scales

Unique Aspects of MPAS Code: Registry, Pools, and Logging

MPAS
Model for Prediction Across Scales

Introduction

When one actually looks inside the MPAS-Atmosphere model, there are several features that can be confusing without proper background:

1. The MPAS "Registry"
2. Pools (less aquatic and fun than you might imagine…)
3. Logging mechanism

This talk is very software-oriented (perhaps boring?), but the ideas will be important for upcoming talks about adding new diagnostics and passive tracers in MPAS!

The MPAS *Registry* File

A central component of all MPAS "cores" is the *Registry* file.

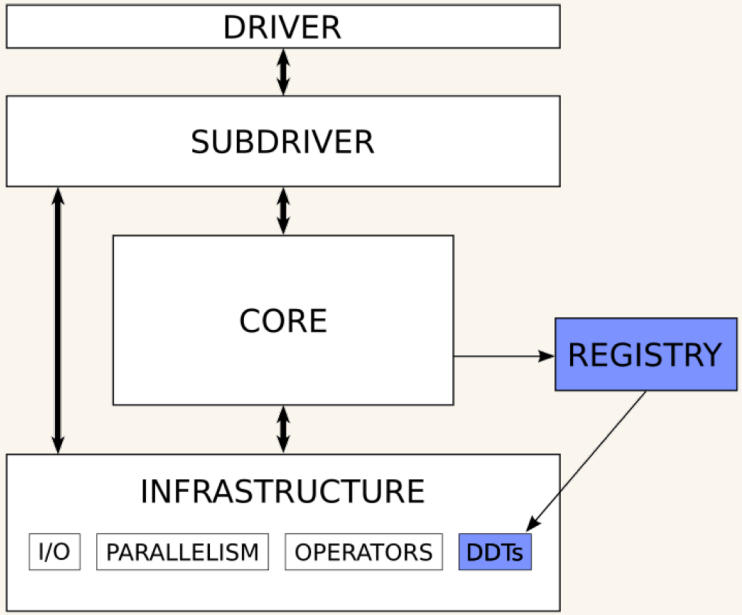
- An idea borrowed from the Weather Research and Forecasting (WRF) model

Motivation:

We wanted to avoid manually writing copy-and-paste code every time we added a new variable or namelist option in MPAS

- Allocation/deallocation
- Addition of fields to data structures
- I/O

The Registry mechanism parses the `Registry.xml` file and writes this code for us!



MPAS
Model for Prediction Across Scales

The MPAS *Registry* File

One could argue that through appropriately designed data structures and functions, we shouldn't have needed to write copy-and-paste code in the first place (and that's true!), but…

By having all namelist options and fields defined in a single XML file, we can automatically generate ~65 pages of documentation!

The MPAS *Registry* File

There are four primary constructs that can are defined in the `Registry.xml` file for an MPAS "core"

1) Namelist options

```
&damping
    config_zd = 22000.0
/
```

This namelist option is defined by this entry in the Registry.xml file

```
<nml_record name="damping" in_defaults="true">
        <nml_option name="config_zd" type="real"
             default_value="22000.0"
             units="m"
             description="Height MSL to begin w-damping profile"
             possible_values="Positive real values"/>
</nml_record>
```

The MPAS *Registry* File

There are four primary constructs that can are defined in the `Registry.xml` file for an MPAS "core"

2) Dimensions

```
<dims>
        <dim name="nCells"
             description="The number of Voronoi cells in the primal mesh"/>

        <dim name="nVertLevels"
             description="The number of atmospheric layers"/>
</dims>
```

The MPAS *Registry* File

There are four primary constructs that can are defined in the `Registry.xml` file for an MPAS "core"

3) Variables

```
<var_struct name="mesh" time_levs="1">

        <!-- horizontal grid information -->
        <var name="latCell" type="real" dimensions="nCells"
             units="rad"
             description="Latitude of cells"/>

        <var name="lonCell" type="real" dimensions="nCells"
             units="rad"
             description="Longitude of cells"/>
</var_struct>
```

NB: the dimensions of variables must themselves be defined in the Registry.xml file as in the previous slide

The MPAS *Registry* File

There are four primary constructs that can are defined in the `Registry.xml` file for an MPAS "core"

4) *Default* I/O streams

```
<streams>
        <stream name="input"
                type="input"
                filename_template="x1.40962.init.nc"
                input_interval="initial_only"
                immutable="true">

                <var name="latCell"/>
                <var name="lonCell"/>
                ...
        </stream>
</streams>
```

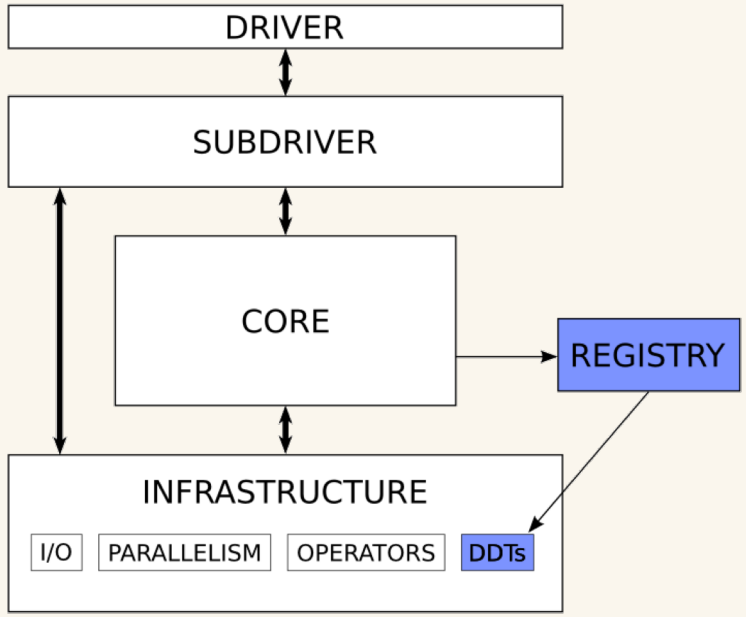
NB: as described in an earlier talk, additional streams can always be defined at run-time in the "streams" file

The MPAS *Registry* File

How and when does the Registry impact MPAS?

During compilation, there is a build step that parses the `Registry.xml` file and generates Fortran code that is included by an MPAS "core"

- Changing the `Registry.xml` file requires recompilation of MPAS!




```
MPAS-Model/
    src/
        core_atmosphere/
            diagnostics/
            dynamics/
            inc/
            physics/
            Registry.xml
            utils/
```

Automatically generated Fortran code goes in the `inc/` directory

- *About 23,300 lines of code for MPAS-Atmosphere v6.1*

MPAS Pools

Closely related to the Registry in MPAS are “pools”

- These are best explained with a little historical perspective…

Right: A picture of a pool from Wikipedia. About the only thing that MPAS pools have in common with this one is that one can add items to a pool and take items from the pool.

MPAS Pools: some history

When we started to develop MPAS, we wanted a way to write more abstract and more maintainable code.

Consider a function, below, for computing dynamics tendencies:

```
subroutine compute_dyn_tend(theta_m, rho_zz, u, w, zgrid, &
                            areaCell, dcEdge, tend_u, tend_rho, &
                            tend_w)

    real, dimension(:,:), intent(in) :: theta_m
    real, dimension(:,:), intent(in) :: rho_zz
    real, dimension(:,:), intent(in) :: u
    real, dimension(:,:), intent(in) :: w
    real, dimension(:,:), intent(in) :: zgrid
    real, dimension(:,:), intent(in) :: areaCell
    real, dimension(:,:), intent(in) :: dcEdge
    real, dimension(:,:), intent(out) :: tend_u
... And so on ...
```

MPAS Pools: some history

With some code comments, it's not hard to figure out what this routine does

- But if any new inputs or outputs are needed, those need to be individually declared and added to the arg list

We wanted to be able to write code like this:

```
subroutine compute_dyn_tend(stateFields, meshInfo, tendencyFields)

    type(state_t), intent(in) :: stateFields
    type(mesh_t), intent(in) :: meshInfo
    type(tend_t), intent(out) :: tendencyFields
```

Where, e.g., `state_t` could be defined as:

```
type state_t
    real, dimension(nVertLevels, nCells) :: theta_m
    real, dimension(nVertLevels, nCells) :: rho_zz
    real, dimension(nVertLevels, nEdges) :: u
    ! ... And so on ...
end type state_t
```

MPAS Pools: some history

The problem comes when we have two different MPAS *cores* (e.g., MPAS-Atmosphere and MPAS-Ocean) that need to define the `state_t` type differently

MPAS-Atmosphere needs this:

```
type state_t
    real, dimension(nVertLevels, nCells) :: theta_m
    real, dimension(nVertLevels, nCells) :: rho_zz
    real, dimension(nVertLevels, nEdges) :: u
    ! ... And so on ...
end type state_t
```

While MPAS-Ocean needs this:

```
type state_t
    real, dimension(nVertLevels, nCells) :: salinity
    real, dimension(nCells) :: SSH
    real, dimension(nVertLevels, nCells) :: layerThickness
    ! ... And so on ...
end type state_t
```

MPAS Pools: some history

Our solution to this problem in MPAS was to develop a generic, dynamic data type called a "pool", which:

- Can be instantiated multiple times
- Can have different fields added to each instance

Now, we can do something like this:

```
type(mpas_pool_type), pointer :: AtmStatePool
type(mpas_field_2d_real_type), pointer :: theta_m_ptr

allocate(AtmStatePool)

call mpas_pool_add_field(AtmStatePool, 'theta_m', theta_m_ptr)
```

Note: the exact type names above don't match the actual MPAS code… I just used names that are closer to what we probably should have chosen in the first place…

MPAS Pools

Whereas before we could access members of types like this:

```
type(state_t) :: State

write(0,*) maxval(State % theta_m)
```

We now have to access members of pools like this:

```
type(mpas_pool_type) :: State

real, dimension(:,:), pointer :: theta_m_ptr

call mpas_pool_get_array(State, 'theta_m', theta_m_ptr)

write(0,*) maxval(theta_m_ptr)
```

Pools are a little more cumbersome to use, but they allow us to write rich infrastructure to perform operations on entire groups of fields!

MPAS Pools

The dimensions, fields, and namelist options that we define in a `Registry.xml` file can all be accessed through pools:

- Dimensions:

```
call mpas_pool_get_dimension(AtmState, 'nCells', nCells)
```

- Fields:

```
call mpas_pool_get_array(AtmState, 'theta_m', theta_m)
```

- Namelist options:

```
call mpas_pool_get_config(Configs, 'config_dt', dt)
```

There are a few more details, but these are the essential ideas…

The MPAS Logging Mechanism

Naturally, we'd like to write out messages as the model runs

- *An informal survey revealed that many of us use print statements as our primary means of debugging!*

What happens if we do something like the following in parallel code (both MPI and OpenMP)?

```
subroutine RHS(arg1, arg2, arg3)
    integer, intent(in) :: arg1, arg2
    integer, intent(out) :: arg3

    write(0,*) arg1, arg2, arg3
```

The MPAS Logging Mechanism

In MPAS v5.3 and earlier, our solution was to cleverly redirect `stdout` and `stderr` to log files named `log.XXXX.out` and `log.XXXX.err`

However, this is less than ideal when several different MPAS components (e.g., ocean, land ice, sea ice) are running together in the same coupled Earth-system model!

MPAS v6.0 introduced a completely new mechanism for logging messages during model execution

The MPAS Logging Mechanism

The standard way for logging a message in MPAS uses the `mpas_log` module's `mpas_log_write(…)` routine

- Each MPAS core writes log messages to a file named `log.<CORE>.0000.out`
- `mpas_log_write(…)` handles tagging of messages with threadID for messages logged from threaded code regions

```
subroutine RHS(arg1, arg2, arg3)
    use mpas_log, only : mpas_log_write

    integer, intent(in) :: arg1, arg2
    integer, intent(out) :: arg3

    call mpas_log_write('Hello from the RHS routine')
```

The MPAS Logging Mechanism: printing variables

Variables can be included in messages with placeholders `$i`, `$r`, or `$l` for integers, reals, or logicals

- The variables to substitute for these placeholders are specified with optional arguments `intArgs`, `realArgs`, and `logicArgs`

```
subroutine RHS(arg1, arg2, arg3)
    use mpas_log, only : mpas_log_write

    integer, intent(in) :: arg1, arg2
    integer, intent(out) :: arg3

    call mpas_log_write('Inputs are $i and $i',   &
                        intArgs=(/arg1, arg2/))
```

The MPAS Logging Mechanism: message types

There are four types of messages that can be written:

1. Regular messages
2. Warnings
3. Error messages
4. *Critical* error messages – *writing one of these will halt the model*!

The type of a log message is specified with the optional argument `messageType`, which can be:

- `MPAS_LOG_OUT` – the default, which doesn't need to be specified
- `MPAS_LOG_WARN`
- `MPAS_LOG_ERR`
- `MPAS_LOG_CRIT`

The MPAS Logging Mechanism: printing variables

An example of `messageType`:

```
subroutine RHS(arg1, arg2, arg3)
    use mpas_log, only : mpas_log_write, MPAS_LOG_ERR

    integer, intent(in) :: arg1, arg2
    integer, intent(out) :: arg3

    if (arg1 < 0) .or. arg2 < 0) then
        call mpas_log_write('Both input args must be >0',&
                            messageType=MPAS_LOG_ERR)
    end if
```

MPAS
Model for Prediction Across Scales

The MPAS Logging Mechanism: message types

The counts of each message type are summarized at the end of model execution (whether that's a successful run or a failed run)

```
-----------------------------------------
Total log messages printed:
   Output messages =                   213
   Warning messages =                    3
   Error messages =                      4
   Critical error messages =             1
-----------------------------------------
```

Logging of errors and critical errors will trigger the creation of a `log.<core>.<processorID>.err` file with the message, e.g.,

```
-rw-r--r-- 1 duda mmm 10260 Jul 28 12:32 log.atmosphere.0000.out
-rw-r--r-- 1 duda mmm   628 Jul 28 12:32 log.atmosphere.0001.err
-rw-r--r-- 1 duda mmm   628 Jul 28 12:32 log.atmosphere.0003.err
```

Above: MPI tasks 1 and 3 both encountered errors, which can be found in the log.atmosphere.0001.err and log.atmosphere.0003.err files.

Summary

The MPAS code is fairly plain Fortran 95/2003, and the key unique features are:

- The Registry, where all fields, dimensions, and run-time options are defined
- Dynamic data structures called “pools”
- A logging mechanism to deal with the complexities of writing messages from parallel executables that may contain more than one MPAS *core*

Having an understanding of these is essential to successfully making changes or additions to the MPAS code!