MPAS Developers Guide

MPAS Development Team

April 19, 2013

Contents

Chapter 1

Summary

This document is intended to describe general development practices within the MPAS project. The information contained should be read prior to starting a project within the MPAS framework. Instructions can be used by MPAS core developers, or external developers. Notes specific to external developers are made where relevant.

1.1 Becoming a MPAS Developer

The preferred approach to contributing source code to the MPAS-Dev repositories is via a fork of the release repository. If the preferred approach is not workable, developer access to the MPAS-Dev GitHub project needs to be approved by core maintainers. Prospective developers should send the following information to the relevant core maintainers.

- What is your GitHub user name?
  (can be acquired at github)
- Can your work be completed in a fork of the release repository?
- If no, please explain the reason.
- What core do you intend to develop?
- What work do you intend to contribute to MPAS?

Chapter 2

Repository Descriptions

The MPAS project contains several repositories. Some of the repositories are private for development efforts while some are public for users and releases. These repositories are described below.

2.1 Main Development/Release

The main MPAS repositories are described in this section. Users interested in implementing a new feature only need to interest themselves in section 2.1.3.

2.1.1 MPAS – Private Repository

This main MPAS development repository is located here. This repository will be where the majority of MPAS development occurs. Instructions on using it to add a feature are below in section 3.2. This is a private repository for development efforts.

2.1.2 MPAS-Legacy – Private Repository

The MPAS-Legacy repository is located here. It provides a linear history of the previous SVN repository's trunk/mpas directory. It is provided only for debugging purposes and no development should/will be carried out using it.

2.1.3 MPAS-Release – Public Repository

The main MPAS release repository is located here. Users should refer to this repository for bug fixes, issues, and major releases for MPAS.

Users who wish to implement a new feature, and have that feature merged into the MPAS repository should follow the instructions on code requirements and repository layout as an MPAS developer would. Forks should be made of this release repository for this type of development rather than the MPAS developer repository. New features will have to undergo a code review before any merge onto the release repository.

2.1.4 Layouts

The MPAS, MPAS-Release, and MPAS-Legacy repositories are all laid out as follows.

```
MPAS
|-Makefile
\-src
  |-driver
  |-registry
  |-framework
  |-operators
  |-external
  |-inc
  |-core_*
  \-Makefile
```

Although there are more files/directories than are listed these are the relevant directories for source code modifications.

The src directory contains all source code related to MPAS, whether it is shared or not.

Shared parts of MPAS belong in either the driver, external, framework, operators, or registry directories.

registry contains the parsing code for Registry. Registry is used for easy modification of variables within a specific MPAS core. Each core has it's own Registry.xml file which defines a list of dimensions, namelist options, variables, and other information specific to that core. On compile time, the registry parser (*parse*) is built. This parser then parses the Registry.xml file and generates some Fortran code, which is stored in src/inc. It is then included at various places within framework to define all of the necessary things. The code generated from registry defines the domain structure for a core, and routines associated with building domain. Domain will be discussed in section 4.1.

driver contains general code for building the MPAS executable, and the general structure of how an MPAS executable looks.

external contains external code to MPAS that is used within MPAS, i.e. ESMF time keeping routines.

framework contains shared code related to the framework of MPAS. This includes the I/O layer, communication routines, and definitions of the data types.

operators contains shared code used for computing specific quantities on an MPAS grid. This includes radial basis function interpolation.

inc is an empty staging directory that registry fills at compile time.

Non-shared parts of MPAS belong under the remaining directories. As an example, the core_sw directory represents the shallow water dynamic core within the MPAS framework. Other directories named core_ are either dynamical cores, or parts of other dynamical cores.

Most developers will only work under their specific core directory, and should not modify code in another core without permission.

2.2 MPAS-Testing – Private Repository

The MPAS-Testing repository is located here. This repository will contain software to setup and run MPAS test cases.

2.2.1 Layout

The MPAS-Testing repository is laid out as follows:

```
MPAS-Testing
|--sw (Test Cases for Shallow water core)
|--ocean (Test Cases for Ocean core)
|--atmosphere (Test Cases for Non-Hydrostatic Atmospheric core)
\--shared (Test Cases for shared parts of MPAS)
```

2.3 MPAS-Documents – Private Repository

The MPAS-Documents repository is located here. This repository will contain all MPAS related documents. This includes design documents, users guides, developers guides, etc.

2.3.1 Layout

The MPAS-Documents repository is laid out as follows:

```
MPAS–Documents
|--design_documents
|   |--shared (Design documents for project in shared directory)
|   |--sw (Design documents for shallow water core)
|   |--ocean (Design documents for ocean core)
|   |--atmosphere (Design documents for Non–hydrostatic atmosphere core)
|--users_guide
|   |--shared (Shared parts of users guide)
|   |--sw (Shallow water specific parts of users guide)
|   |--ocean (Ocean specific parts of users guide)
|   |--atmosphere (Non–hydrostatic Atmosphere specific parts of users guide
\--developers_guide (This document)
```

2.4 MPAS-Tools – Private Repository

The MPAS-Tools repository is located here. This repository will contain all MPAS related tools.

2.4.1 Layout

The layout of the MPAS-Tools repository is as follows:

```
MPAS–Tools
|--analysis (All tools related to analysis)
|--grid_gen (All tools related to grid generation)
|--python_scripts (All general purpose python scripts)
\--visualization (All visualization tools)
```

To promote reuse of previously developed tools, tools won't be stored in core specific directories. This way, a developer is encouraged to browse other tools and see if a feature is implemented elsewhere.

2.5 MPAS-Data – Public Repository

The MPAS-Data repository is located here. This repository will contain data required to run specific MPAS models. This data does not include input files related to specific runs, it is intended to house general data that doesn't change between runs.

2.5.1 Layout

The layout of the MPAS-Data repository is as follows:

```
MPAS–Data
|--framework (Data required for parts of framework)
|--operators (Data required for parts of operators)
\--core_* (Data required for specific cores)
```

Chapter 3

Development

The repositories are all hosted on github. The typical life-cycle of a project is as follows:

1. Create a design document for the project.
   (e.g. section 3.1)
2. Visit appropriate repository website.
   (e.g. Release Repository)
3. Create a fork of the repository.
   (e.g. section 3.2)
4. Locally clone the newly created fork.
   (e.g. git clone https://fork/url)
5. Create a branch within the fork, for the new feature or bug fix.
   See section 3.4 for a description of our branching strategy.
   (e.g. git checkout -b new_branch)
6. Develop branch.
   (e.g. touch src/framework/work_complete.F && git commit -a)
7. Push complete branch to remote fork.
   (e.g. git push -u origin new_branch)
8. Submit a pull request to merge branch on fork to main repository master.
   (e.g. section 3.3)

Projects don't have to follow this example verbatim, but this at least gives a general overview of the process. Some details related to this life-cycle will be described in the following sections.

Code level requirements are described in chapter 4.

3.1 Design Documents

Design documents are recommended for projects that contribute significant changes to either a core or to MPAS in general. They should clearly describe the justification for the project, and the changes and impacts of the project. Core maintainers reserve the right to deny a pull request that lacks a design document.

Developers can refer to DDT Redesign and Implicit Vertical Mixing (MPAS-Ocean) for examples of design documents that have been previously completed.

3.2 Forks

A fork is a user specific copy of another repository. Forks can only be made from repositories a user has pull access to. Permissions on a fork remain the same as the original repository (e.g. pull teams still have pull access over the fork), with the exception that the user creating the fork gains elevated admin permissions over the newly created fork.

A fork can be thought of exactly like a branch. Except it lives in a separate repository, and contains it's own private history. As forks remember where they came from, they can still be merged onto the repository they were created from.

The github fork guide can be used to learn how to create a fork of a repository.

The user who created a fork can easily delete the fork, without fear of destroying the original repository.

3.3 Pull Requests

A pull request represents several things in our development process. At it's basis, a pull request is a merge request. It describes a branch that a developer wishes to have merged onto some alternate branch or repository. In addition to a request for a merge, it fully describes the changes involved

in the merge, along with allowing for a full review of the merge prior to the actual merge.

Other developers can review and comment on pull requests. A pull request will be assigned to the relevant core maintainer.

Github provides guides for creating a pull request, using a pull request, and merging a pull request. Please refer to these guides for more instructions on pull requests.

3.4 Branch Strategy

As a developer, most work will be completed in a branch. This branch can be one of several types of branches. To give an overview of the current branching strategy, please see this document.

The name of a branch should be descriptive and tell where the feature addition will be. For example, if the branch is intended to implement multiple blocks, the majority of its work would take place in framework. A good name for the branch would then be framework/multiple_blocks.

Feature branches should only be created from the develop branch. This allows a cleaner work flow when planning releases.

Chapter 4

Development guidelines

As developers of MPAS, we attempt to make the code look as uniform as we can across the entire code-base. In order to enforce this, there are a set of guidelines developers should follow.

- Each core has a name, and an abbreviation. For example, the shallow water core is called sw and it's abbreviation is sw, but the ocean core is called ocean and it's abbreviation is ocn.
- All subroutines should be named in a manner which prevents namespace conflicts.
  Shared functions/subroutines are simply named mpas_subroutine_name.
  Core specific functions/subroutines are named mpas_abbrev_subroutine_name (where abbrev is replaced with the cores abbreviation).
  e.g. mpas_atm_time_integration
- Subroutine names should all be lower case, with underscores in place of spaces (e.g. see above).
- Variable names should be mixed case (e.g. cellsOnCell rather than cells_on_cell).
- In general, variable names should be self-descriptive (e.g. nCells rather than n).
- Subroutines and modules should be appropriately documented. Shared portions of MPAS code use doxygen comments, but core developers are free to decide what method of documenting they prefer.
- Development of shared parts of MPAS need reviews from multiple core maintainers prior to a merge.

- Development within a core should be approved by other core developers before being merged into that core.
- Development within a core should follow the practices of that core's developer group, for documentation etc.
- Core related testing is the responsibility of that core's maintainers/developers.

Core maintainers need to approve changes before the changes appear on the shared repository. Core maintainers are also responsible for reviewing changes to shared code that affect multiple cores.

4.1 General Code Introduction

The MPAS framework makes extensive use of derived data types. A lot of these derived data types are defined in src/framework/mpas_grid_types.F, but some of the data types are not defined until build time. Registry is used to define both fields and structures. A field is the lowest level of data with in the MPAS framework, and contains a description of the field along with the actual field data. The field types that are allowed are defined in the previously mentioned file, and Registry makes use of these type definitions.

A structure is a larger grouping of fields, and dimensions. Structures can have multiple time levels, or a single time level. A structure can be used to group fields and define a part of the model, e.g. the mesh definition.

Structures are grouped into blocks. A block can be thought of as all relevant information for a particular part of your domain. The entire domain is subdivided horizontally into blocks. Within the model, blocks are stored in a linked list.

The largest structure is the domain, which defines all blocks that are part of a particular MPI processes computational domain.

The following diagram is provided as a visual aid for the layout of data within the MPAS framework. It is not exhaustive, so please refer to any core specific references to get a more complete list of how derived data types are laid out in the specific core.

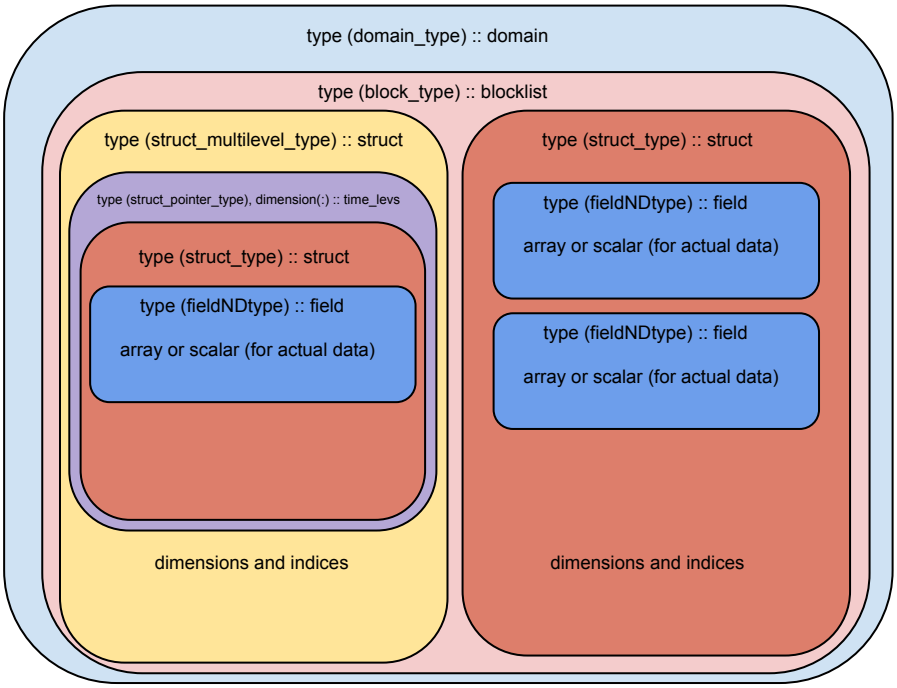

Figure 4.1: Visual diagram of MPAS derived data type layout.

In the actual MPAS code, this diagram converts to the following lines.

```
domain % blocklist % struct1 % field1 % array
domain % blocklist % struct1 % field2 % scalar

domain % blocklist % struct2 % time_levs(1) % struct2 % field1 % array
domain % blocklist % struct2 % time_levs(1) % struct2 % field2 % scalar
```

As the blocklist structure is a linked list of blocks, one can iterate over the list of blocks in the following manner.

```
type (block_type), pointer :: block_ptr

block_ptr => domain % blocklist
do (while(associated(block_ptr))
        ... do stuff on block ...
        block_ptr => block_ptr % next
end do
```

In order to create new fields or structures, a developer needs to modify Registry.xml for the particular core. The definition of Registry.xml is stored in src/registry/Registry.xsd. This schema file can also be used to validate a Registry.xml file using any XML validator.

4.2 Parallelization Strategy

Currently within MPAS, the only parallelization strategy is MPI. This section is intended to be a basic introduction to the use of MPI within MPAS.

Within MPAS, the horizontal dimensions are partitioned into *blocks*. A block can be thought of as a self describing portion of the global domain. In the event MPAS is run with a single processor, only one block is used which encompasses the entire domain. These partitions are generated by using an external tool *metis*, that partitions a graph file.

Upon initialization of MPAS, the graph partition file is read in. Each processor is then responsible for constructing it's blocks. A processor can be in charge of one or multiple blocks, where "in charge" means responsible for determining the correct answer in. On a single MPI task, all blocks are stored in a linked list as described in section 4.1. Within the block are lists of fields. Each of these fields has some number of halos. A halo is a boundary of ghost elements to ensure the correct answer on actual owned elements. Figures 4.2 and 4.3 show how halo layers might be defined in a simulation with two cell halo layers. Edge and Vertex halo layers share a definition.

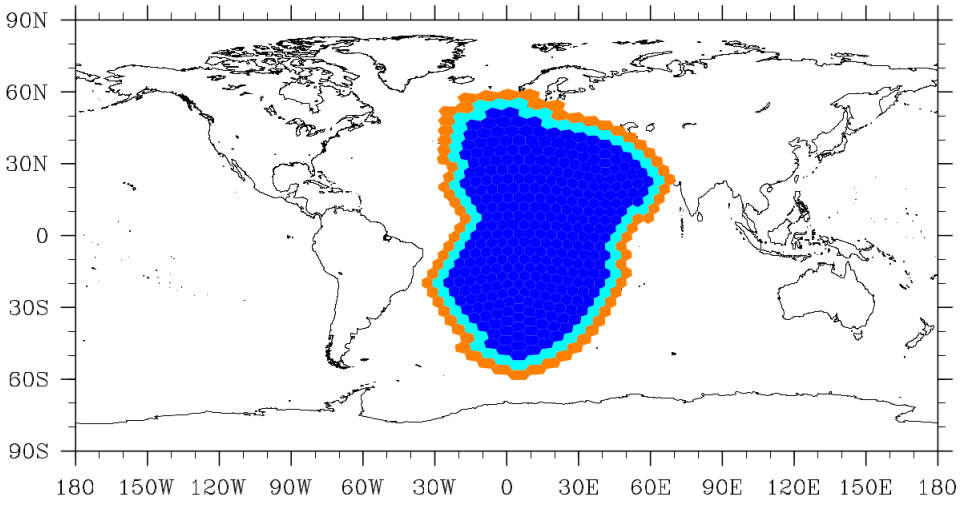

Figure 4.2: The dark blue cells are the blocks owned cells. The light blue are the first halo layer cells and the orange are the second halo layer cells.

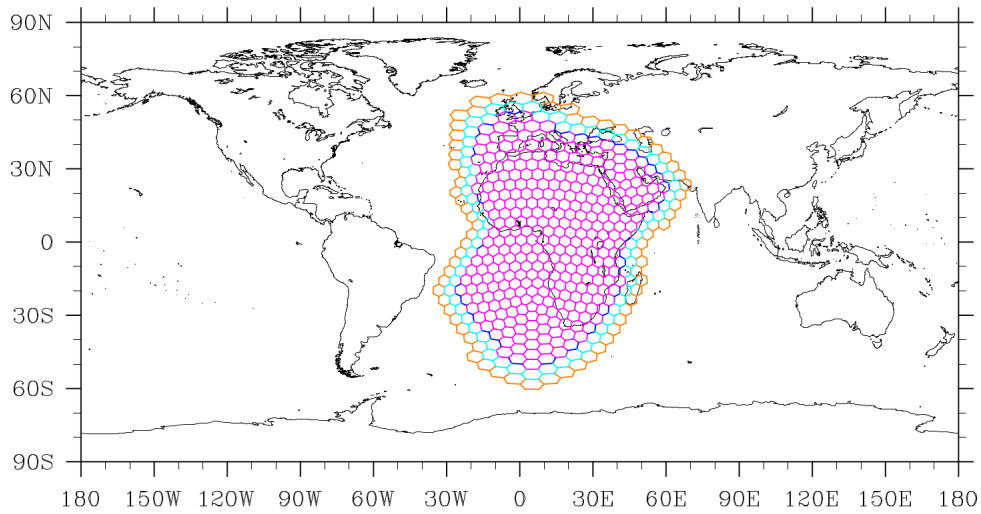

Figure 4.3: The pink edges are the blocks owned edges. The dark blue edges on the pink perimeter are the first halo layer edges, the light blue are the second halo layer edges, and the orange are the third halo layer edges.

Cells make up the primary mesh within MPAS. Halos for cells are described below:

- 0-Halo: All "owned" cells;
- 1-Halo: All cells that border 0-Halo cells but are not 0-Halo cells;
- 2-Halo: All cells that border 1-Halo cells but do not belong to a lesser halo;
- 3-Halo: All cells that border 2-Halo cells but do not belong to a lesser halo;

Cells can have an arbitrary number of halos, in which case this incremental definition continues until the last halo layer.

Edges and Vertices have a slightly different definition and always include an additional layer.

- 0-Halo: All "owned" edges/vertices;
- 1-Halo: All edges/vertices that border 0-Halo cells that are not 0-Halo edges/vertices;
- 2-Halo: All edges/vertices that border 1-Halo cells that are not 1-Halo edges/vertices;
- 3-Halo: All edges/vertices that border 2-halo cells that are not 2-Halo edges/vertices;
- ...

As with cells, edges/vertices past the 1 halo use the same definition until the last halo layer.

MPI communication occurs between these halo layers. One block needs to send some of it's 0-halo to other blocks, and it needs to receive other block's 0-halo elements to put into it's halo layers. This communication occurs through exchange lists, which are built during initialization. An exchange list describes MPI sends, MPI receives, and local copies.

A local copy occurs when two blocks need to communicate and are owned by the same MPI task. In this case, the exchange lists describes where data can be found in the owning blocks list, and where it should go in the receiving blocks list.

A MPI send exchange list describes what processor the communication needs to occur with. It also describes what elements need to be read from

the owning blocks list, and where those elements should be placed in the sending buffer.

A MPI receive exchange list describes what processor the communication needs to occur with. It also describes how to unpack data from the buffer into the halo layers.

Halo exchange routines are provided in src/framework/mpas_dmpar.F. These halo exchange routines internally loop over all blocks in the list of blocks, starting from the block that was passed in. Because of this, a developer should always pass in the first block's field (i.e. domain % blocklist % struct % field) to these routines, and should never put a halo exchange within a block loop. A halo exchange within a block loop will only work properly if there is only a single block per processor.

A generic interface for halo exchanges of all fields is provided as follows:

```
mpas_dmpar_exch_halo_field(field, haloLayersIn)
```

The argument haloLayersIn is optional, and allows specification of which halo layers should be communicated. This allows the developer more control over message size.

Currently there is no standard way of implementing OpenMP in MPAS. None of the shared code has any OpenMP directives in it, but several developers are investigating the best method of implementing OpenMP in MPAS.

Chapter 5

Addition of a new core

Some developers might want to add a new core to the main MPAS repository, without having a core maintainer defined yet. The steps of adding a new core are largely the same as adding a new feature.

The new core is developed in a fork of the relevant repository. After the new core is ready to be merged into the main repository, a pull request is submitted. This pull request can be assigned to any other core maintainer, and should be reviewed by at least two core maintainers.

The other core maintainers will review the core, and ensure it meets the requirements set forth in the developers guide. After the new core is approved to be merged onto the developers repository, a core maintainer may be designated to handle pull requests related to the new core.

Chapter 6

Core maintainers

Each core has a list of maintainers. These lists should be relatively short, and include developers with write access to the main developers repository. Core maintainers will be responsible for pull requests into their core, and possibly shared portions of MPAS.

Current core maintainers are as follows:

- Non-Hydrostatic Atmosphere:
  - Michael Duda
  - Bill Skamarock
- Ocean
  - Doug Jacobsen
  - Mark Petersen
  - Todd Ringler
- Shallow Water
  - All Other Core Maintainers