Scratch Variables: Requirements and Design

MPAS Development Team

March 8, 2013

Contents

Chapter 1

Summary

This document describes the requirements and implementation of scratch variables, along with the general use guidelines.

Chapter 2

Requirements

2.1 Requirement: Non-Persistent Global Fields

Date last modified: 03/07/13
Contributors: (Doug Jacobsen)

Scratch variables are defined as non-persistent fields. Currently these types are defined, allocated, and deallocated at the module level. As OpenMP implementations have yet been determined for core/shared code, scratch varaibles are created to allow heap defined arrays that can be public to multiple threads.

Chapter 3

Design and Implementation

3.1 Implementation: Non-Persistent Global Fields

Date last modified: 03/07/13
Contributors: (Doug Jacobsen)

Currently in Registry, one defines a persistent variable in a line similar to the following:

```
var persistent integer cellsOnEdge ( TWO nEdges ) 0 iro cellsOnEdge mesh -
```

where the keyword is persistent. This means that MPAS is responsible for allocating and deallocating this field at the beginning and end of a simulation. The alternative would be changing the word persisntent to scratch.

This modification would mean that the developer is responsible for allocating and deallocating the field at the beginning and end of use (or a subroutine).

In order to accomplish this, MPAS needs to provide a developer with subroutines that can allocate and deallocate these scratch fields. MPAS will provide generic interfaces for allocation and deallocation as follows:

```
subroutine mpas_allocate_scratch_field(f, single_block_in)
subroutine mpas_deallocate_scratch_field(f, single_block_in)
```

These generic interfaces lie on top of more specific interfaces such as:

```
subroutine mpas_allocate_scratch_field4d_real(f, single_block_in)
```

Each of these subroutines takes one required argument and one optional argument. The optional argument is single_block_in, which tells the routine

if you want to allocate/deallocate a single block, or all blocks in a blocklist.

The subroutines will then allocate or deallocate all field pointers requested (either a single block or all blocks), and return.

Once a scratch field is allocated, it can be used exactly as a persisntent field. However, one caveat is that a scratch variable can't be included in any I/O streams, as the I/O layer has to assume since it's a scratch variable that it will not be allocated when it has access to it.

Chapter 4

Use Guidelines

Scratch variables are intended to be used as temporary storage space. The alternative is persisntent variables which are intended to be used as permanent storage space. Because scratch is inheriently temporary, a developer should assume they are responsible for all memory management with scratch varaibles.

This means a general requirement is that the developer should ensure they call the allocate and deallocate routines at the correct places within the code. The definition of correct largely depends on the application, but a good rule of thumb is that correct is immediately before and immediately after the variable is required, and incorrect would be anywhere else in the core. Typically scratch variables should be allocated and deallocated within the subroutine that makes use of them. Allocation would happen at the beginning of the subroutine, and deallocation would happen at the end of the subroutine. The easiest use case would be replacing module level variables that have a dimension of nElements with a scratch variable, everything is almost identical between the two.

Scratch variables should have useful names, similarly to persisntent variables, that represent the quantity (although it is temporary) that should be stored in them.

If a scratch variable is going to be used at a level below where it's allocated (within another subroutine) that subroutine should expect it as an argument, and assume that it's already allocated upon calling. In general, a scratch data structure can be passed around (similarly to grid or mesh) that contains all scratch variables. The subroutine can then use only the relevent variables from that data structure.

Scratch variables can't be used for simple 1D arrays that *should* be

thread private. Examples of this are 1D arrays that are dimensioned by nVertLevels. The scratch counterpart of this 1D array would be a 2D array that is dimensioned nVertLevels by nElements. However, 1D arrays that are dimensioned by nElements are fine to use in scratch variables.

In general, there's no reason one can't manually allocate the array pointer for a scratch variable and make use of it exactly as a module level variable. This should be discouraged as it doesn't follow the general use guidelines of traditional variables, and could cause other developers problems when using that scratch variable.