The Interface between the MPAS Land Ice Dynamic Core and the Velocity Solver: Requirements and Design

MPAS Development Team

January 5, 2012

Contents

Chapter 1

Summary

The MPAS system is developing a new land ice dynamic core. The geometry of a given region is modeled by a grid of cells which are organized into vertical layers or sheets. The values of physical quantities of interest, called state variables, are to be computed at specific locations within each cell, and over a sequence of time values. Thus a typical computation involves some initial setup, associated with defining the grid, followed by a sequence of timestep calculations which compute estimates for the state variables at the next time.

A particular state variable needed at each time step is the instantaneous velocity field. There are a number of approaches to computing this quantity, and the MPAS land ice development team has decided that the computation should be carried out by a separate procedure, here called the *velocity solver* module.

It is desirable that the internal details of the velocity solver module be left free to the developers. This is so because:

- it may be desirable to provide a choice of solvers corresponding to different models;
- velocity solvers may be designed by independent groups;
- it may be preferable to apply the finite element method to model the velocity equations.

Thus, the land ice dynamic core, as it carries out a time step, must be prepared to transfer information about the current state to the velocity solver module, and then to accept back the velocity field as it has been computed.

Given these considerations, the appropriate approach is to carefully define an interface that controls the flow of state information, geometry, and other conditions to the velocity solver, and the return of information that defines the velocity field. If this interface is carefully and clearly defined, then the velocity solver itself can be considered a "black box". As long as designers carry out the required calculations, and respect the communication interface, they will then be free to implement the computation according to their interests.

This document is specifies the details of the interfaces needed in order for the MPAS land ice dynamic core to successfully interact with a velocity solver module. This entails the information needed to specify the calculation, and the format of the desired results. Furthermore, it lays out an explicit interface, that is, a formal function call, with a parameter list, whose types, dimensions, and meanings are determined.

This information allows developers of the MPAS dynamic core and those working on velocity solvers to proceed independently on code design, without having to wait for each other. In particular, a group working on one side of the interface can test their code by supplying a simple dummy function to stand on the other side of the interface and supply "canned data".

A first step towards judging the success of the interface is to run the full program on one side, with a dummy function on the other, which simply goes through the motions of computing data, but actually simply sends a standard set of test data. Thus, the velocity solver can be partially tested by sending it data from a dummy version of MPAS, while the MPAS land ice dynamic core can be tested by having a dummy velocity solver return canned velocities back through the interface.

Chapter 2

Requirements

2.1 Requirement: MPAS Supplies Initialization Input Needed by Velocity Solver

Date last modified: 2011/09/15
Contributors: John Burkardt, Mauro Perego

At first, it was thought that the grid interface would be called only once, and therefore it could be used to pass initialization data as well as the grid information. However, it seems practical to assume that a single run of the program might include the necessity to modify the grid several times. In this case, it is useful to supply an additional interface routine which is guaranteed to be used only once, and which passes one-time initialization data.

2.2 Requirement: MPAS Supplies Grid and Geometry Input Needed by Velocity Solver

Date last modified: 2011/06/07
Contributors: John Burkardt, Mauro Perego

MPAS stores the geometry, grid, boundary conditions, and state variables for the entire computation. The velocity solver needs some of this information once, at the beginning of the computation, in order to set up its own geometry model. Thus, it is required that MPAS supplies to the velocity solver the grid and geometry information, before any time steps are

computed, using an agreed interface.

2.3 Requirement: MPAS Supplies State Variable Input Needed by Velocity Solver

Date last modified: 2011/06/07
Contributors: John Burkardt, Mauro Perego

MPAS stores and updates state variables for the entire computation. The velocity solver will need to receive updated information on the current values of the state variables and boundary conditions in order to correctly set up the calculation for the corresponding velocity at the new time. Thus, it is required that MPAS supplies to the velocity solver the changing state information at each time step, using an agreed interface.

2.4 Requirement: Velocity Solver Returns Output Needed by MPAS

Date last modified: 2011/06/07
Contributors: John Burkardt, Mauro Perego

During each time step, once the velocity solver has determined the velocity field, this information must be returned to MPAS. The velocity solver can also determine certain related quantities derived from the velocity information, and needed by MPAS. This information must be supplied in a suitable format, and at appropriate geometric positions, as required by MPAS. Thus, it is required that the velocity solver returns information to MPAS during each time step, using an agreed interface.

Chapter 3

Algorithmic Formulations

3.1 Design Solution: Identification of Initialization Input Needed by Velocity Solver

Date last modified: 2011/09/15
Contributors: John Burkardt, Mauro Perego

At the moment, the only data likely to be passed in this way is the identifier for the MPI communicator.

3.2 Design Solution: Identification of Grid and Geometry Input Needed by Velocity Solver

Date last modified: 2011/06/07
Contributors: John Burkardt, Mauro Perego

The velocity solver needs the location of the cell centers of the MPAS mesh, the number of layers, and various connectivity arrays in order to recreate the discretized geometry. Assuming the basic structure of the mesh does not vary over the time steps, then this information should only need to be transferred once.

3.3 Design Solution: Identification of State Variable Input Needed by Velocity Solver

Date last modified: 2011/06/01
Contributors: John Burkardt, Mauro Perego

In order to set up the equations for the velocity, the velocity solver needs, on each time step, the state variables and boundary condition information that affect this computation.

3.4 Design Solution: Identification of Velocity Solver Output

Date last modified: 2011/06/01
Contributors: John Burkardt, Mauro Perego

Once the velocity solver has computed a solution, MPAS will require velocity values, as well as certain other derived quantities, at specific locations in the mesh. The velocity solver may carry out interpolation or other suitable procedures in order to produce these values at the desired locations.

Chapter 4

Design and Implementation

4.1 Implementation: Initialization Input from MPAS to Velocity Solver

Date last modified: 2011/09/15
Contributors: John Burkardt, Mauro Perego

The velocity solver needs certain information, one time only, as initialization. This information includes:

- *comm*, the identifier for the MPI Communicator **MPI_COMM_WORLD**, as established when MPAS invokes MPI; the velocity solver needs this communicator in order to make use of MPI as MPAS has initialized it;

4.2 Implementation: Grid and Geometry Input from MPAS to Velocity Solver

Date last modified: 2011/06/02
Contributors: John Burkardt, Mauro Perego

It is assumed that the geometry of the MPAS grid is fixed throughout the computation.

The velocity solver needs mesh information from MPAS in order to construct the grid. This information includes:

- *(xCell,yCell,zCell)*, the coordinates of the MPAS cell centers; under

discussion is the idea of using 2D coordinates from a polar stereographic projection;

- *cellsOnVertex*, the indices of cells incident on a vertex;
- *indexToVertexID*, the tags for vertices;
- *verticesOnEdge*, indices of the two vertex endpoints of a cell edge;
- *nVertLevels*, the number of vertical layers.

4.3 Implementation: State Variable Input from MPAS to Velocity Solver

Date last modified: 2011/06/07
Contributors: John Burkardt, Mauro Perego

As each new time step begins, MPAS updates certain variables that define the state of the system, and which the velocity solver needs in order to properly set up the system defining the velocities. These items include:

- *temperature*, the temperature at each cell center, for each layer;
- *thickness*, the thickness, associated with each cell center;
- *elevation*, the elevation, associated with each cell center;
- *?*, the sigma layers *(no name suggested; not clear on the meaning)*;
- *beta*, the sliding coefficient, and other basal parameters, associated with each cell center.
- *flowfactor*, the flow factor $A(T)$ of each layer in each cell;
- *emptyCell*, a logical variable that indicates whether a cell is empty *(this is a suggestion for how to handle the "empty cell" issue.)*.

4.4 Implementation: Output from Velocity Solver to MPAS

Date last modified: 2011/06/07
Contributors: John Burkardt, Mauro Perego

The velocity solver determines the velocity vector field. MPAS requires the velocity solver to return certain velocity components at selected points, as well as related information:

- *u, v*, normal and tangential velocities at the midpoint of each edge face of each layer;
- *heat dissipation*, the tensor product of the full stress tensor and the strain tensor integrated over each layer of each cell;
- *viscosity*, possibly, for diagnostics, the viscosity of each cell;
- *?*, possibly, for diagnostics, terms of the stress and strain rate tensors; *(no name specified; not clear on details.)*;

4.5 Implementation: Format of the Initialization Interface

Date last modified: 2011/09/15
Contributors: John Burkardt, Mauro Perego

The routine **velocity_solver_init()** is intended to be used to pass such information between MPAS and the velocity solver. From the MPAS side, the routine would behave according to the following declarations:

```
subroutine velocity_solver_init ( comm )

  integer, intent ( in ) :: comm
```

From the C++ side, the routine would be declared as follows:

```
void velocity_solver_init ( int *comm );
```

4.6 Implementation: Format of the Grid and Geometry Interface

Date last modified: 2011/12/15
Contributors: John Burkardt, Mauro Perego

Before the time step calculation begins, the MPAS routine **land_ice_init** in **module_land_ice_core** sets up the geometry and the grid. In order for

the velocity solver to update its internal version of the geometric grid and state variables to correspond to changes made by MPAS, it is necessary that certain grid and state variable data be transferred. This is accomplished by providing four interface routines, to be called by the MPAS routine **land_ice_init**. Each interface routine looks like a FORTRAN subroutine to MPAS, and can be documented as such. In fact, however, the underlying operations are carried out by a C++ function.

The primary interface routine is **set_velocity_solver_grid()**. This function builds the two-dimensional triangular grid (already partitioned among the processors). The grid will keep only the vertices selected by the array `verticesMask`. The function **extrude_velocity_solver_3d_grid()** is called to construct the vertical structured three-dimensional grid, starting from the two-dimensional grid. After the 3D grid has been built, the solver to be employed is determined by a call to either **initialize_L1L2_solver** or **initialize_FO_solver**, which primarily build the necessary finite element spaces.

In brief, the interface functions have the FORTRAN90 and C++ forms:

subroutine set_velocity_solver_grid ( )	void set_velocity_solver_grid_ ( )
subroutine extrude_velocity_solver_3d_grid()	void extrude_velocity_solver_3d_grid_()
subroutine initialize_FO_solver()	void initialize_FO_solver_()
subroutine initialize_L1L2_solver()	void initialize_L1L2_solver_()

These routines are interfaces between a FORTRAN90 main program and a set of C++ subsidiary functions. This means that it is necessary to be able to think of them, abstractly, as being written in either language. Here we produce the corresponding declarations that would be associated with a FORTRAN90 or C++ version of each of the interface functions.

For the FORTRAN90 version, this information would have the form:

```
subroutine set_velocity_solver_grid(nCells, nEdges, nVertices, &
  nCellsSolve, nEdgesSolve, nVerticesSolve, &
  cellsOnEdge, cellsOnVertex, verticesOnCell, verticesOnEdge,&
  verticesMask, xCell, yCell, zCell &
  sendCellsArray, recvCellsArray, sendEdgesArray, &
  recvEdgesArray, sendVerticesArray, recvVerticesArray)

  integer, intent ( in ) :: nCells
  integer, intent ( in ) :: nEdges
  integer, intent ( in ) :: nVertices
  integer, intent ( in ) :: nCellsSolve
  integer, intent ( in ) :: nEdgesSolve
```

```
  integer, intent ( in ) :: nVerticesSolve

  integer, intent ( in ) :: cellsOnEdge(2,nEdges)
  integer, intent ( in ) :: cellsOnVertex(3,nVertices)
  integer, intent ( in ) :: verticesOnCell(6,nCells)
  integer, intent ( in ) :: verticesOnEdge(2,nEdges)

  logical, intent ( in ) :: verticesMask(nVertices)

  double precision, intent ( in ) :: xCell(nCells)
  double precision, intent ( in ) :: yCell(nCells)
  double precision, intent ( in ) :: zCell(nCells)

  integer, intent ( in ) :: sendCellsArray(variableSize)
  integer, intent ( in ) :: recvCellsArray(variableSize)
  integer, intent ( in ) :: sendEdgesArray(variableSize)
  integer, intent ( in ) :: recvEdgesArray(variableSize)
  integer, intent ( in ) :: sendVerticesArray(variableSize)
  integer, intent ( in ) :: recvVerticesArray(variableSize)

subroutine extrude_velocity_solver_3d_grid(nVertLevels, thickness, &
  elevation)

  integer, intent ( in ) :: nVertLevels

  double precision, intent ( in ) :: thickness(nCells)
  double precision, intent ( in ) :: elevation(nCells)

subroutine initialize_FO_solver()

subroutine initialize_L1L2_solver()
```

In the above, the arrays send*Xxx*Array and recv*Xxx*Array store the connectivity between the processors. They have the following structure.
`sendCellsArray(1)` stores the size of the vector.
`sendCellsArray(2)` stores the ID of the processor `P_i1` to send data to
`sendCellsArray(3)` stores the number of Ids `n_Pi1` to be sent
`sendCellsArray(4:3+n_Pi1)` store the Ids to be sent to `P_i1`
`sendCellsArray(4+n_Pi1)` stores the ID of the processor `P_i2` to send data to and so on.

The C++ declarations of the interface routines have the following form:

```
void set_velocity_solver_grid_(int const * nCells_F,
  int const * nEdges_F, int const * nVertices_F,
  int const * nCellsSolve_F,
  int const * nEdgesSolve_F, int const * nVerticesSolve_F,
  int const * cellsOnEdge_F, int const * cellsOnVertex_F,
  int const * verticesOnCell_F, int const * verticesOnEdge_F,
  int const * verticesMask_F, double const * xCell_F,
  double const * yCell_F, double const * zCell_F,
  double const * thickness_F,  double const * elevation_F,
  int const * sendCellsArray_F, int const * recvCellsArray_F,
  int const * sendEdgesArray_F, int const * recvEdgesArray_F,
  int const * sendVerticesArray_F, int const * recvVerticesArray_F)

void extrude_velocity_solver_3d_grid_(int const * nVertLevels_F,
  double const * thickness_F,  double const * elevation_F)

void initialize_FO_solver_()

void initialize_L1L2_solver()
```

One consideration for the C++ implementation is that FORTRAN90 double-dimensioned arrays are stored in column-major format. This means that the C++ code, which gets a pointer to the first element of each array, needs to index the array information by following the FORTRAN90 conventions.

4.7 Implementation: Format of the State Variable Interface

Date last modified: 2011/15/12
Contributors: John Burkardt, Mauro Perego

Each time integration step of the land ice calculation is carried out in **module_time_integration**. Within this module, subroutine **timestep()** controls the computation of data associated with the new time by calling the routine **rk4()**, which uses a fourth-order Runge Kutta method to advance the solution data in time.

Since the velocity solver, instead of **rk4()**, will be computing the new velocities, then **rk4()** must be modified so that it no longer attempts to compute velocity updates, and **timestep()** must now call both **rk4()** and

the velocity solver. The generic format of this portion of **timestep()** might look something like this:

```
if (trim(config_time_integration) == 'RK4') then
   call rk4 ( domain, dt )
else
   write(0,*) 'Unknown time integration option ' &
   // trim ( config_time_integration )
   write(0,*) 'Currently, only ''RK4'' is supported.'
   stop
end if

call velocity_solver_XXX ( elevation, thickness, beta, temperature,
  nVertLevels, u, v, heatIntegral, viscosity )
```

Function `velocity_solver_XXX` stands for one of `velocity_solver_SIA`, `velocity_solver_SSA`, `velocity_solver_L1L2`,`velocity_solver_FO` and possibly for other solvers (like Stokes). It computes the ice velocity. Output variables are the normal and tangential velocities on the midpoint of edges, heat dissipation integrated over a cell and the viscosity on each cell.

The Fortran declaration of the function `velocity_solver_XXX` is:

```
subroutine velocity_solver_XXX ( elevation, thickness, beta, &
  temperature, nVertLevels, u, v, heatIntegral, viscosity )

  double precision, intent ( in ) :: elevation(nCells)
  double precision, intent ( in ) :: thickness(nCells)
  double precision, intent ( in ) :: beta(nCells)
  double precision, intent ( in ) :: temperature(nCells, nVertLevels)

  integer, intent ( in ) :: nVertLevels

  double precision, intent ( out ) :: heatIntegral(nCells,nVertLevels)
  double precision, intent ( out ) :: viscosity(nCells,nVertLevels)
  double precision, intent ( out ) :: u(nEdges,nVertLevels)
  double precision, intent ( out ) :: v(nEdges,nVertLevels)
```

The corresponding C++ declaration is

```
void velocity_solver_XXX_ ( double const * elevation_F,
  double const * thickness_F, double const * beta_F,
```

```
double const * temperature_F, double const * nVertLevels_F,
double const * u_F, double const * v_F,
double const * heatIntegral_F, double const * viscosity_F ).
```

Type	Name	Dimension	Definition
double precision	beta	(nCells)	the sliding coefficient.
integer	cellsOnVertex	(3, nVertices)	the indices of cells incident on a vertex.
integer	comm	1	id for **MPI_COMM_WORLD**.
double precision	elevation	(nCells)	the elevation at each cell center.
double precision	thickness	(nCells)	layer thickness at cell center.
logical	verticesMask	(nVertices)	true if vertex should be kept.
double precision	flowfactor	(nCells,nVertLevels)	the flow factor at cell centers.
double precision	heatIntegral	(nCells,nVertLevels)	heat dissipation integrated over each cell.
integer	nCells	1	the number of cells, (unknowns + ghosts).
integer	nCellsSolve	1	the number of cells, (unknowns only).
integer	nEdges	1	the number of cell edges, (unknowns + ghosts).
integer	nEdgesSolve	1	the number of cell edges, (unknowns only).
integer	nVertices	1	the number of cell vertices, (unknowns + ghosts).
integer	nVerticesSolve	1	the number of cell vertices, (unknowns only).
integer	nVertLevels	1	the number of vertical layers, (typically 10 or 11).
double precision	temperature	(nCells,nVertLevels)	temperature at cell centers.
double precision	u	(nEdges,nVertLevels)	the normal velocity at the midpoint of each edge face of each layer.
double precision	v	(nEdges,nVertLevels)	the tangential velocity at the midpoint of each edge face of each layer.
integer	cellsOnEdge	(2,nEdges)	the pair of cells that share the edge.
integer	cellsOnVertex	(3,nVertices)	the three cells that share the vertex.
integer	verticesOnCell	(6,nCells)	the six vertices of the cell
integer	verticesOnEdge	(2,nEdges)	the pair of vertices that are the endpoints of the cell edge.
double precision	viscosity	(nCells,nVertLevels)	the viscosity of each cell.
double precision	xCell	(nCells)	the **x** coordinate of the cell center on the unit sphere.
double precision	yCell	(nCells)	the **y** coordinate of the cell center on the unit sphere.
double precision	zCell	(nCells)	the **z** coordinate of the cell center on the unit sphere.

Table 4.1: Glossary of MPAS/Velocity Solver Interface Variables

Chapter 5

Testing

5.1 Testing and Validation: Passing Dummy Data Through the Interface

Date last modified: 2011/06/01
Contributors: John Burkardt, Mauro Perego

As soon as a reasonably detailed interface has been agreed upon, it will be possible for developers on either side of the interface to test their code, and the interface, by creating a dummy version of the program that would normally be running on the other side of the interface.

Thus, the MPAS developers will be able to write a simple procedure for producing "plausible" velocity values and related quantities, and to call that procedure through the interface.

Similarly, developers of the velocity solver functions can set up main programs that pass in a test case geometry and time evolution data.

Such trial runs enable each developer to make simple test runs of their code in cases where the other "half" of the code is not yet available; it also allows developers to spot situations in which the interface does not include certain information that is required for the calculations to proceed properly.

5.1.1 Test of steady state velocity solver on greenland

In file `test_steady_state_greenland` the netCDF grid **gis_20km.180511.nc** is read and a structured triangular mesh of greenland is obtained, calling the function `set_velocity_solver_grid`. Then a SIA problem is solved prescribing no-slip boundary conditions on the bedrock and setting a con-

stant flow rate A. Similarly one can solve the L1L2 model, but in this case a robin bedrock boundary condition must be available.