MPAS I/O

February 24, 2012

Contents

Chapter 1

Introduction

In order to support multiple blocks of cells per MPI task, there are a number of development issues that need to be addressed:

1. Update/extend the fundamental derived types in mpas_grid_types.F. In order for other parts of the infrastructure to handle multiple blocks per task in a clean way, we'll need to be able to pass a head pointer to a field into a routine, and have that routine loop through all blocks for that field, with information about which cells/edges/vertices in that field need to be communicated.

2. Decide on a new MPAS I/O abstraction layer, which will provide a high-level interface to the PIO layer for the rest of MPAS. This layer should work with blocks of fields, and make it possible to define an arbitrary set of I/O streams at run-time.

3. Add a new module to parse a run-time I/O configuration file that will describe which fields are read or written to each of the I/O streams that a user requests via the file. This module will make calls to the new MPAS I/O layer to register the requested fields for I/O in the requested streams.

4. Update the mpas_dmpar module to support communication operations on multiple blocks per task. This will likely involve revising the internal data structures used to define communication of cells between tasks, and also require revisions to the public interface routines themselves.

5. Modify the block_decomp module to enable a task to get a list of cells in more than one block that it is to be the owner of. Implemented in the simplest way, there could simply be a namelist option to specify how many blocks each task should own, and the block_decomp module could look for a graph.info.part.n file, with n=num_blocks_per_task*num_tasks, and assign blocks k, 2k, 3k, ..., num_blocks_per_task*k to task k.

This document addresses the requirements and design of a new MPAS I/O layer (Item 2, above) that will provide much-needed functionality, including the ability to perform I/O on multiple blocks per MPI task, to perform input and output in parallel, and to allow for an arbitrary number of I/O streams (as well arbitrary set of fields in each of those streams) to be defined by the user.

Chapter 2

Requirements

In order to meet current I/O needs, and to provide flexibility for future extension, the new I/O layer in MPAS must meet the following requirements.

- The I/O interface must allow the user to define sets of fields (constituting a "stream") that are read or written as a group from/to a file at a common time. The I/O times, as well as the fields in the stream, are decided on a per-stream basis.
- It must be possible to designate each stream as either an input stream or an output stream.
- There must be no artificial limit (i.e., aside from memory limits) to the number of streams that can be concurrently in use.
- The I/O interface must allow the user to choose which I/O "format" (among any that are implemented by the I/O layer) to use on a per-stream basis. A format refers to the file format and method used to write the file, e.g., netCDF, pNetCDF or binary via MPI-IO. At a minimum, the I/O layer must implement both serial netCDF and pNetCDF.
- The I/O interface must support the ability to read and write variable attributes and global attributes.
- The I/O interface must support fields with multiple blocks on an MPI task.
- For an identical field, it must be possible to produce identical file output through the I/O layer regardless of the MPI task count, the distribution of blocks between MPI tasks, or the distribution of cells between blocks.

Chapter 3

Design

In this chapter, the design of the MPAS I/O layer is described. First, a summary of implementation issues is presented to help in understanding the constraints placed on the design; a major consideration is the fact that, although partitions of the MPAS SCVT meshes are currently computed off-line, to be read at model start-up, MPAS should ultimately be capable of computing this partition 'on-line', that is, at run-time. Having discussed the major design constraints, the interface to the new I/O layer is then presented in detail.

3.1 Issues and constraints

3.1.1 Model bootstrapping

The process of reading fields at the beginning of an MPAS model run is inherently tied to the process of partitioning the SCVT mesh at run-time. This follows from the fact that cell-connectivity information must be used in the generation of the partitions, yet this information resides in the very file to be read as the *cellsOnCell* field. Further, in order to partition the edges and vertices of the mesh, information on the connections between the edges, vertices, and cells is needed, and this information is also stored in a file as *edgesOnCell*, *cellsOnEdge*, *verticesOnCell*, and *cellsOnVertex*. In short, cell-based fields are needed in order to partition the SCVT mesh so that these and other cell-based fields can be read in parallel onto their (computed) computational partitions (i.e., blocks), and similarly for edge-based and vertex-based fields.

The current procedure for dealing with these issues relies on a bootstrapping procedure, in which:

1. The total number of cells, edges, and vertices in the mesh are read from the input file from the dimensions *nCells*, *nEdges*, and *nVertices*.

2. A contiguous range of cell, edge, and vertex global indices is assigned to each task, e.g., cells nint(mpi_rank * nCells / mpi_size)+1 through nint((mpi_rank+1) * nCells / mpi_size).

3. Each I/O task reads its range of global indices for the fields *indexToCellID*, *indexToEdgeID*, *indexToVertexID*, *nEdgesOnCell*, *cellsOnCell*, *edgesOnCell*, *verticesOnCell*, *cellsOnEdge*, and *cellsOnVertex*.

4. A partitioning of the SCVT mesh is requested from the block_decomp module, given a distributed description of the mesh connectivity based on the *cellsOnCell* field (distributed across all I/O tasks); currently, this partitioning is read from a `graph.info` file.

5. The *indexToCellID*, *nEdgesOnCell*, and *cellsOnCell* fields are re-distributed so that each of the tasks owns the global indices of these fields that were assigned to it by the partitioning of the mesh.

6. Halos are constructed for the cells; a halo consists of all of the cells referenced in the *cellsOnCell* array that are not in the indexToCellID array.

7. The *edgesOnCell* and *verticesOnCell* fields are re-distributed so that each task has these fields for all cells in its block(s), including halo cells.

8. Each task constructs a list of edges and vertices adjacent to cells in the blocks(s) owned by that task.

9. The *cellsOnEdge* and *cellsOnVertex* fields are re-distributed so that each task has these fields for all edges and cells in its block(s).

10. The edges and vertices in each block are divided into owned and halo edges and vertices based on the *cellsOnEdge* and *cellsOnVertex* fields; an edge *iEdge* is owned iff *cellsOnEdge(1, iEdge)* is an owned cell, and a vertex *iVtx* is owned iff *cellsOnVertex(1, iVtx)* is an owned cell.

11. Knowing how many (and which) cells, edges, and vertices are in each block (as well as which are owned and which are ghost), block data structures are allocated by each task.

12. Fields are then read in parallel and re-distributed among the tasks into the field arrays of the block data structures on each task.

In this procedure, it is important to note that every compute task is also an I/O task. This will not be true in future, where the I/O tasks may be a subset of the MPI tasks (or possibly even a disjoint set of tasks).

3.1.2 Super-arrays

Currently, the registry-generated I/O code in the mpas_io_input and mpas_io_output modules handles the details of packing and unpacking individual constituent arrays from super-arrays; for example, in the atmosphere models, the fields *qv*, *qc*, and *qr* exist as individual fields in input and output files, but are packaged together in a "super-array" of one higher dimension in the model, namely, as the array *scalars*. To support multiple blocks per MPI task, the I/O system will most naturally work with the derived data types for fields and blocks, since these types contain links between blocks on the same MPI task. However, there is currently no information available in the field types to indicate whether the field is a super-array, and, if so, the names of its constituents. The new I/O system could rely on the MPAS registry to generate code internal to the module to handle super-arrays, but such an approach would not easily facilitate run-time determination of the number of scalar constituents in a model, nor would it lead to completely general I/O code. In the new I/O layer, it would be preferable to have no registry-generated internal code (internal to the module), and to require that the field types be extended to contain super-array information, so that the I/O layer would be presented with all necessary information to pack or unpack these super-arrays.

3.2 High-level approach

Conceptually, the bootstrapping part of the input procedure described in the preceding section (steps 1 – 11) is independent of the particular I/O streams that will be later used by the model to read initial conditions, periodically update boundary conditions, and write history or restart files (step 12); also, the bootsrapping only needs to be performed once at model start-up, regardless of the number of streams used in the model. Toward the goal of presenting a simple stream-oriented interface to the model developer and hiding the details of getting, assigning, and allocating blocks, yet minimizing the amount of code to be written and maintained, we propose to split the MPAS I/O interface into two parts. One part will provide low-level routines to open and close files, read and write arbitrary index ranges of individual arrays, read and write attributes, etc., with the level of abstraction similar to that of the netCDF or PIO interfaces. The second, high-level part of the I/O interface will provide routines for creating a stream, adding MPAS field-types to the stream, and reading or writing the stream. The high-level routines will work with the derived data types for fields and blocks defined in the mpas_grid_types module, and their functionality will be built on the functionality provided by the low-level interface, which will be used principally during bootstrapping.

One important consequence of allowing the user to define an arbitrary set of streams, in light of the need for a bootstrapping procedure, is that some file containing the information needed by the bootstrapping procedure must always be designated by the user. The responsibility for meeting this requirement will be taken on by the run-time I/O specification module, described in Item 3 of the Introduction.

3.3 High-level interface description

The high-level interface is expected to be the primary interface to MPAS I/O for the user (i.e., model developer), assuming the bootstrapping procedure needed to partition the global mesh and allocate blocks has been done. Using the interface described in this section, a typical set of calls might look something like the following.

```
call MPAS_io_init(dminfo, 16, 32, ierr)      ! From the "low-level" interface
call MPAS_createStream(init, 'x1.10242.init.nc', &
                       MPAS_STREAM_PNETCDF, MPAS_STREAM_INPUT, 0, ierr)
call MPAS_streamAddField(init, theta, ierr)
call MPAS_streamAddField(init, u, ierr)
call MPAS_streamAddField(init, w, ierr)
call MPAS_streamAddField(init, qv, ierr)
call MPAS_streamAddField(init, qc, ierr)
call MPAS_streamAddField(init, qr, ierr)
call MPAS_readStream(init, 1, ierr)
call MPAS_readStreamAtt(init, 'on_a_sphere', isSphericalGrid, ierr)
call MPAS_readStreamAtt(init, 'sphere_radius', radius, ierr)
call MPAS_closeStream(init, ierr)
call MPAS_io_finalize(ierr)                  ! From the "low-level" interface
```

One point not obvious from the interface description concerns streams that have a mix of time-varying and time-invariant fields. For such streams, the time-invariant fields will be read only on the first call to MPAS_readStream for the stream; subsequent calls to MPAS_readStream will only

read the specified time frame for time-varying fields. Similarly, calls to MPAS_writeStream will only write time-invariant fields on the first call for the stream, or whenever the specified number of frames per file has been exceeded and a new output file must be created; thus, for output streams, every file created from that stream will contain a copy of the time-invariant fields.

Although there are routines for reading and writing global attributes, no analogous routines exist in the high-level interface for variable attributes. In the proposed design, the set of variable attributes is fixed as those attributes in the `io_info` type described in the mpas_grid_types module; the values of these attributes are automatically written and read when a stream is written or read. The rationale behind this decision is that, while global attributes may frequently be changed to reflect new information that needs to be carried around with a dataset, the variable attributes are more likely to be fixed to meet, e.g., CF metadata conventions.

---

subroutine

MPAS_createStream(stream, filename, io_format, io_direction, frames_per_file, ierr)

Creates a new I/O stream, to which fields can be added before reading or writing the stream. For input streams, the number of frames per file must be 0 or 1, and for output streams the number of frames per file can be any number ≥ 0; if frames_per_file > 0, the first timestamp in the file will be inserted automatically into the filename based on the value of mesh%xtime.

Input

character (len=*) :: filename — *The name of the file to which the stream will be connected; if io_direction is MPAS_STREAM_INPUT, filename must refer to an existing file*

integer :: io_format — *The form of the stream, either MPAS_STREAM_NETCDF or MPAS_STREAM_PNETCDF*

integer :: io_direction — *Whether the stream is an input or output stream, specified with either of the constants MPAS_STREAM_INPUT, MPAS_STREAM_OUTPUT*

integer :: frames_per_file — *For time-varying fields, the maximum number of time frames that can exist in a file for the stream; 0 indicates an unlimited number of frames*

Output

type(MPAS_Stream_type) :: stream — *The newly created I/O stream*

integer, optional :: ierr — *The return error code*

---

subroutine

MPAS_streamAddField(stream, field, ierr)

Adds an MPAS field type to the set of fields in the stream; the field can be any of the field types defined in the mpas_grid_types module, e.g., field2DReal.

Input

type(MPAS_Stream_type) :: stream — *An MPAS stream previously created with a call to MPAS_createStream*

type(field2DReal) :: field — *The field to be added to the stream*

Output

integer, optional :: ierr — *The return error code*

---

subroutine

MPAS_readStream(stream, frame, ierr)

Reads all fields associated with the stream; for time-varying fields, the field will be read at the time-frame specified by the frame argument.

Input

type(MPAS_Stream_type) :: stream — *The I/O stream to read*

integer :: frame — *For time-varying fields, the time frame to be read; ignored for time-invariant fields*

Output

integer, optional :: ierr — *The return error code*

---

subroutine

MPAS_writeStream(stream, frame, ierr)

Writes all fields associated with the stream; for time-varying fields, the field will be written at the time-frame specified by the frame argument.

Input

type(MPAS_Stream_type) :: stream — *The I/O stream to write*

integer :: frame — *For time-varying fields, the time frame to be written*

Output

integer, optional :: ierr — *The return error code*

---

subroutine

MPAS_readStreamAtt(stream, attName, attValue, ierr)

Reads a global attribute from the stream; the type of the attribute in the stream must match the type of the attValue argument.

Input

type(MPAS_Stream_type) :: stream — *The I/O stream from which the attribute will be read*

character (len=*) :: attName — *The name of the attribute to read*

Output

integer :: attValue — *The value of the attribute*

integer, optional :: ierr — *The return error code*

---

subroutine

MPAS_writeStreamAtt(stream, attName, attValue, ierr)

Writes a global attribute to the stream, with the type of the attribute determined by the type of the attValue argument.

Input

type(MPAS_Stream_type) :: stream — *The I/O stream to which the attribute will be written*

character (len=*) :: attName — *The name of the attribute to write*

integer :: attValue — *The attribute value to be written*

Output

integer, optional :: ierr — *The return error code*

---

subroutine

MPAS_closeStream(stream, ierr)

Closes an I/O stream.

Input

type(MPAS_Stream_type) :: stream — *The I/O stream to be closed*

Output

integer, optional :: ierr — *The return error code*

3.4 Low-level interface description

The main purpose of the low-level MPAS I/O interface is to support the bootstrapping procedure at model start-up and to support the functionality of the high-level I/O interface (i.e., to allow the high-level interface to be implemented using a package-independent interface). Of course, if the user requires a greater level of control over the reading or writing of a file, the low-level interface could in principle be used directly without tying the resulting user code to a particular external package (e.g., PIO or netCDF).

---

subroutine

MPAS_io_init(dminfo, io_task_count, io_task_stride, ierr)

Initializes the MPAS I/O layer; this routine must be called once by every task before any subsequent calls to MPAS I/O routines are made.

Input

type(dm_info) :: dminfo — *The dminfo structure returned by the mpas_dmpar module*

integer :: io_task_count — *The number of I/O tasks to use when reading and writing streams*

integer :: io_task_stride — *The stride between I/O tasks*

Output

integer, optional :: ierr — *The return error code*

---

function

MPAS_io_open(filename, mode, ioformat, ierr)

Opens a file, either for reading or writing, using the specified file-level format.

Return value

A handle (of type MPAS_IO_Handle_type) to the opened file to be used in subsequent calls to the MPAS low-level I/O layer.

Input

character (len=*) :: filename — *The name of the file to open*

integer :: mode — *Either of the constants MPAS_IO_READ or MPAS_IO_WRITE, specifying whether the file is to be opened for reading or writing*

integer :: ioformat — *The format of the file, either MPAS_IO_NETCDF or MPAS_IO_PNETCDF*

Output

integer, optional :: ierr — *The return error code*

---

subroutine

MPAS_io_inq_dim(handle, dimname, dimsize, ierr)

Returns the value of a dimension in a file opened for reading.

Input

type(MPAS_IO_Handle_type) :: handle — *An MPAS file handle*

character (len=*) :: dimname — *The name of the dimension*

Output

integer :: dimsize — *The size of the dimension*

integer, optional :: ierr — *The return error code*

---

subroutine

MPAS_io_def_dim(handle, dimname, dimsize, ierr)

Sets the value of a dimension in a file opened for writing.

Input

type(MPAS_IO_Handle_type) :: handle — *An MPAS file handle*

character (len=*) :: dimname — *The name of the dimension*

integer :: dimsize — *The size of the dimension; the constant MPAS_IO_UNLIMITED_DIM indicates an unlimited (record) dimension; only one unlimited dimension may be defined in a file*

Output

integer, optional :: ierr — *The return error code*

---

subroutine

MPAS_io_inq_var(handle, fieldname, fieldtype, ndims, dimnames, dimsizes, ierr)

Returns information (determined by the optional parameters passed to the routine) about a variable in a file opened for reading.

Input

type(MPAS_IO_Handle_type) :: handle — *An MPAS file handle*

character (len=*) :: fieldname — *The name of the field*

Output

integer, optional :: fieldtype — *The type of the field, identified by one of the module constants MPAS_IO_REAL, MPAS_IO_INTEGER, or MPAS_IO_LOGICAL*

integer, optional :: ndims — *The dimensionality of the field*

character (len=64), dimension(:), pointer, optional :: dimnames — *An array of dimension names for the field, which will be allocated by the routine with size ndims*

integer, dimension(:), pointer, optional :: dimsizes — *An array of dimension sizes for the field, which will be allocated by the routine with size ndims*

integer, optional :: ierr — *The return error code*

---

subroutine

MPAS_io_def_var(handle, fieldname, fieldtype, dimnames, ierr)

Defines a variable in a file opened for writing. The dimensionality of the field is determined by the size of the dimnames argument.

Input

type(MPAS_IO_Handle_type) :: handle — *An MPAS file handle*

character (len=*) :: fieldname — *The name of the field*

integer :: fieldtype — *The type of the field, identified by one of the module constants MPAS_IO_REAL, MPAS_IO_INTEGER, MPAS_IO_LOGICAL*

character (len=64), dimension(:) :: dimnames — *An array of dimension names, all of which must have been defined previously with calls to MPAS_io_def_dim(), with the size of the array determining the dimensionality of the field*

Output

integer, optional :: ierr — *The return error code*

---

subroutine

MPAS_io_get_var_indices(handle, fieldname, indices, ierr)

Returns the global indices into the decomposed outermost dimension that will be read by the MPI task for the specified field. Each global index must be specified by at most one task.

Input

type(MPAS_IO_Handle_type) :: handle — *An MPAS file handle*

character (len=*) :: fieldname — *The name of the field*

Output

integer, dimension(:), pointer :: indices — *An array giving the global indices that will be read on this task for the field; the routine will allocate the array to match the size of the index set to be returned*

integer, optional :: ierr — *The return error code*

---

subroutine

MPAS_io_set_var_indices(handle, fieldname, indices, ierr)

Sets the global indices into the decomposed outermost dimension that will be read by the MPI task for the specified field. Each global index must be specified by at most one task.

Input

type(MPAS_IO_Handle_type) :: handle — *An MPAS file handle*

character (len=*) :: fieldname — *The name of the field*

integer, dimension(:) :: indices — *An array of global indices to be written by this task for the field*

Output

integer, optional :: ierr — *The return error code*

---

subroutine

MPAS_io_get_var(handle, fieldname, array, ierr)

Reads the part of a field determined by the global indices that were previously specified in a call to MPAS_io_set_var_indices(); the size of the outer-most dimension of the array argument must match the size of the index array passed to a call to MPAS_io_set_var_indices() for the field. This is an overloaded routine, and the type of the array argument must match the type of the field in the file.

Input

type(MPAS_IO_Handle_type) :: handle — *An MPAS file handle*

character (len=*) :: fieldname — *The name of the field*

Output

various types, dimension(:) :: array — *The part of the field to be read by this task*

integer, optional :: ierr — *The return error code*

---

subroutine

MPAS_io_put_var(handle, fieldname, array, ierr)

Writes the part of a field determined by the global indices that were previously specified in a call to MPAS_io_set_var_indices(); the size of the outermost dimension of the array argument must match the size of the index array passed to a call to MPAS_io_set_var_indices() for the field. This is an overloaded routine, and the type of the array argument will determine the type of the field written to the file.

Input

type(MPAS_IO_Handle_type) :: handle — *An MPAS file handle*

character (len=*) :: fieldname — *The name of the field*

various types, dimension(:) :: array — *The part of the field to be written by this task*

Output

integer, optional :: ierr — *The return error code*

---

subroutine

MPAS_io_get_att(handle, attName, attValue, fieldname, ierr)

Returns the value of an attribute from a file. If a fieldname is specified, the attribute is a variable attribute; otherwise, the attribute is a global attribute. This is an overloaded routine, and the type of the attValue argument must match the type of the attribute in the file.

Input

type(MPAS_IO_Handle_type) :: handle — *An MPAS file handle*

character (len=*) :: attName — *The name of the attribute*

character (len=*), optional :: fieldname — *If present, the name of the field to which the attribute is attached*

Output

various types :: attValue — *The value of the attribute*

integer, optional :: ierr — *The return error code*

---

subroutine

MPAS_io_put_att(handle, attName, attValue, fieldname, ierr)

Sets the value of an attribute in a file. If a fieldname is specified, the attribute is a variable attribute; otherwise, the attribute is a global attribute. This is an overloaded routine, and the type of the attValue argument will determine the type of the attribute written to the file.

Input

type(MPAS_IO_Handle_type) :: handle — *An MPAS file handle*

character (len=*) :: attName — *The name of the attribute*

various types :: attValue — *The value of the attribute*

character (len=*), optional :: fieldname — *If present, the name of the field for which attName is an attribute*

Output

integer, optional :: ierr — *The return error code*

---

subroutine

MPAS_io_close(handle, ierr)

Closes a file that was previously opened with a call to MPAS_io_open().

Input

type(MPAS_IO_Handle_type) :: handle — *An MPAS file handle*

Output

integer, optional :: ierr — *The return error code*

---

subroutine

MPAS_io_finalize(ierr)

Finalizes the MPAS I/O layer. This routine must be the last MPAS I/O routine called once by every task.

Output

integer, optional :: ierr — *The return error code*

Chapter 4

Implementation

TBD

Chapter 5

Testing

TBD