Revisions to MPAS block decomposition routines

May 2, 2012

Contents

Chapter 1

Introduction

Previously several projects have been worked on with the end goal of supporting multiple blocks. These tasks are described below.

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 related to tying all of these projects together, and allowing MPAS to run with multiple blocks per MPI process.

As an example of what running with multiple blocks means, currently MPAS is allowed to run with 1 block per process. This means prior to running MPAS a decomposition of cells needs to be determined. This decomposition tells each MPI process which cells it owns. So, under the current framework the number of decompositions have to be equal to the number of MPI tasks used to run the job.

After this project is completed, a user should be able to specify the use of a decomposition that does not have an equal number of blocks as the number of processors the job is run with. Typically this would be used to allow more than one block per processor, but could theoretically be used to run having some processors not have any blocks.

The goal with this project is to allow exploration of OpenMP directives, and test performance using different block sizes. Allowing multiple blocks per processor could increase cache reuse and allow a performance increase.

This work is currently being performed under branches/omp_blocks/multiple_blocks and all source code changes can be examined there.

Chapter 2

Requirements

There are significant changes to MPAS' framework that have to be made in order to support multiple blocks. A list of requirements that determine these changes are listed below.

- Block creation must be robust, and handle an arbitrary number of blocks per processor. **MGD: What does it mean for block creation to be 'robust'? Will we support zero blocks per processor?**
- Blocks should be created using the derived data types created in an earlier project, utilizing the field data types. **MGD: Does this imply that simple arrays will not be used in any of the block_decomp routines? If simple arrays will be used in some cases, what will distinguish those cases from those that use field types?**
- Block creation routines should be created with an arbitrary number of halos assumed, although the default is currently two. **MGD: Will we support different numbers of halo layers on a per-field basis, or will we require that all fields and blocks have the same number of halo layers?**
- All block communication routines should be able to handle shared memory copies.
- Exchange list creation should be performed at the block/field level.
- A new module should be setup to handle the management of blocks. **MGD: Is this really a requirement?**

MGD: Generally, it seems to me like a couple of the above are really design issues, rather than requirements. I think the core requirements from above are: support for arbitrary numbers of blocks per task; support for an arbitrary number of halo layers (across all fields and blocks); and communication routines that take advantage of shared-memory copies. Maybe we could add something about having user-level interfaces that minimize the amount of redundant information that must be passed to them, effectively ruling out designs that use simple arrays and exchange lists in user-level interfaces in favor of interfaces that work with derived types.

Chapter 3

Design

Only a small amount of design has been completed thus far. So, all information in this section should be regarded as a work in progress for now.

The current prototyping efforts have determined the following routines which require changes to their infrastructure.

```
mpas_dmpar_alltoall_field
mpas_dmpar_exch_halo_field
mpas_dmpar_get_owner_list
mpas_input_state_for_domain
```

Within the allToAll and exch_halo routines, loops over multiple blocks need to be added where they are not currently in place. Also, shared memory copies within local blocks need to be added.

The whole structure of mpas_dmpar_get_owner_list has to change in order to support multiple blocks. This routine currently builds the exchange lists for a single block and has global communications. In order to handle the creation of exchange lists with multiple blocks, this routine will be rewritten to use the field data types. This way each field will be a linked list, consisting of the fields from each block. The downside to this change, is that now the routine requires the setup of basic block types (really just the blockID number), and block fields (like indexToCellID) prior to calling this. However, this keeps the data types used in the creation of these routines in line with how the rest of MPAS deals with fields.

mpas_input_state_for_domain also has to be modified in order to setup the fields and blocks as required for mpas_dmpar_get_owner_list, and to create multiple blocks. Currently it is writing under the assumption that only one block per process exists, and has that single block hard-coded as the only one that gets created. **MGD: See notes below — I think a lot of the work currently in mpas_input_state_for_domain should be migrated elsewhere.**

In addition to the currently in place changes that need to be made, several routines need to be added. Within the current prototyping work, a routine has been created to determine the all cell indices for a block, including all halo cells. It has been written under the assumption that there could be an arbitrary number of halos. This routine is called

```
mpas_get_halo_cells_and_exchange_lists(dminfo, nHalos,
                indexToCellID_0Halo, nEdgesOnCell_0Halo,
                cellsOnCell_0Halo, indexToCellID_nHalos,
                nEdgesOnCell_nHalos, cellsOnCell_nHalos)
```

MGD: It's not clear to me from the argument list how exchange-list information is returned; perhaps determining the exchange lists would be better handled by a

separate routine anyway, given that we'll need to do it for edge and vertex locations as well, and the routine to get all owned and halo edges and vertices might look slightly different from the one for cells.

At the time of writing this document, this routine can be seen within the src/framework/mpas_block_decomp.F module, but this may change when a new module is created.

One general change that has to be made in order to support these field data types being used in the input stage of MPAS is the addition of a deallocate field routine. This routine would be used to deallocate all fields within a field linked list. It is used when a field is created that's not a member of a block, so calling mpas_deallocated_block would not destroy all the memory associated with the field.

In addition to the changes listed here, routines still need to be determined to create a list of vertices and edges for a block and all it's halos, as well as their respective exchange lists. After the list of cells, vertices, and edges are complete for a block the IO read fields can be called to setup the fields within each block. Finally, the global indices within a block need to be modified to be local indices.

As mentioned in the requirements section, a large portion of these changes might be pushed into a new module. This new module would be written to handle the management of blocks. The proposed name would be mpas_block_manager.

MGD: I think this work might be a good opportunity to better organize the parts of the infrastructure that bring the model from the start of execution up to the point where the final fields and blocks can be allocated and initial fields can be read. So, it might be beneficial to discuss some of the higher-level organization of the modules and their conceptual purposes. Here are a few of the issues that I'd been thinking about:

- **Much of the code in the current mpas_io_input module really belongs elsewhere.**
- **With the new high-level IO layer, the only *real* user-level IO work that needs to be done is to add the attributes and fields that belong to each stream; this is currently done via registry-generated code in mpas_io_input and mpas_io_output. To separate out user-level IO work from the 'bootstrapping' work of determining block decompositions, building halos, and allocating fields and blocks, perhaps we could reduce the work in the mpas_io_input and mpas_io_output to only that of adding fields to streams (this would incidentally work out well for later work on run-time-specifiable IO), and create a new module for the boostrapping work; this new module would utilize the proposed mpas_block_manager module, and could serve as something of a driver for all of the model startup work.**
- **This new startup module would leverage the low-level IO layer for reading in the necessary information for getting an assignment of blocks for each task (remember, we may want to do this on-the-fly in future, and so will need information like cellsOnCell to pass to the routine that returns a block assignment), and would then use the block_decomp module to get an assignment of cells to the blocks owned by the task. It might be good to confine the role of the mpas_block_decomp module to simply that of returning block decompositions, whether this is accomplished by reading graph.info.part.* files or by calling Metis or Scotch libraries. Then, the proposed mpas_block_manager module could be employed to build halo layers. Finally, blocks could be allocated and linked together, and initial conditions could be read via the high-level IO layer.**

- **My preference would be to have all of the direct calls to MPI routines confined to a single module like the current mpas_dmpar.**

Chapter 4

Testing

The end goal from this project is to provide a framework that allows bit-for-bit reproduction of data using an arbitrary combination of blocks and processor numbers.

Using this goal to define a testing strategy implies a good test would be exploring bit-for-bit reproduction of output data using the three following simulations:

- Current trunk simulation run with 8 processors and 8 blocks
- Finished branch simulation run with 8 processors and 8 blocks
- Finished branch simulation run with 1 processor and 8 blocks
- Finished branch simulation run with 2 processors and 8 blocks

If all of these simulations produce bit-for-bit output then the project would be deemed as completed.