Implementing Multiple Blocks within MPAS Framework

May 3, 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, with the reasons for these requirements written below them.

- Block creation must be robust, and handle an arbitrary number of blocks per processor.
- Blocks should be created using the derived data types created in an earlier project, promoting the use of field data types rather than simple arrays.
- Block creation routines should be created with an arbitrary number of halos assumed, although the default is currently two.
- All block communication routines should be able to handle shared memory copies.
- Exchange list creation should be performed at the block/field level.
- Block creation code should be isolated from the rest of MPAS code.

Blocks per processor should be allowed to be any non-negative number, including zero. This could be useful if a user wanted the ability to specify certain processors to do certain tasks, without doing any actual computation work on blocks. Although the user would have to give an explicit block to proc decomposition in order to have this ability used.

In the creation of blocks, field data types should be used in place of simple arrays to promote the use of internal derived data types that are used elsewhere within MPAS. This will allow similar techniques to be identified by developers of cores, and allow a similar work flow with variables and fields within all of MPAS.

Although fields currently are restricted to having two halo layers, at some point in the future we might like to be able to extend halo layers or even have different halo layers on each field. In order to make this task easier to accomplish in the future block creation routines need to be able to create an arbitrary number of halo layers.

When two blocks are neighboring on a single processor shared memory copies could be used for halo exchanges and other sorts of block-to-block communications rather than using MPI send/recv routines.

Exchange lists are limited in their functionality by the fact that the only information they have refer to the other processor/block involved in the communication. For example, if processor 0 owns block 0 and this has to send information from 15 cells to processor 2 block 5 then the send list for block 0 only gives the information on where it has to send the information, while the receive list on

block 5 gives information on where to receive the information from. Because of this, exchange lists have to be created on a per block basis, and linked to a specific block. This way each block knows which cells it's supposed to send/recv/copy to/from another block. Each field and block have their own exchange lists, so the creation of exchange lists should place them within these already existing structures.

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 old allToAll interfaces will have to change in order to accommodate the new field data types. The old interface looks like

```
subroutine mpas_dmpar_alltoall_field1d_real(dminfo, arrayIn,
                          arrayOut, nOwnedList, nNeededList,
                          sendList, recvList)
```

In this case, arrayIn and arrayOut are simple arrays representing 1d real values (there are other interfaces for integers, chars, and multi-dimensional arrays but they are all similar), nOwnedList and nNeededList are integers representing the sizes of arrayIn and arrayOut respectively, and sendList and recvList are exchange lists describing how the data from arrayIn needs to be communicated into arrayOut.

The proposed new interface looks like

```
subroutine mpas_dmpar_alltoall_field1d_integer(dminfo,
                                    fieldIn, fieldOut)
```

Where fieldIn and fieldOut are pointers to the fields that need to be communicated. In this case the exchange lists are stored within the field, or possibly the field % block data structure. Both of the two fields represent a linked list of fields, where each of the fields in this linked list is the field for a given block. Each of the fields in the linked list also has it's own unique exchange lists.

In order to handle allToAll communications, the following pseudo code is used

```
loop over fieldOut list
  loop over recvList for specific field
    initiate mpi_irecv for specific recvList
```

```
  end loop
end loop

loop over fieldIn list
  loop over sendList for specific field
    if sendList % procID == dminfo % my_proc_id
      loop over fieldOut list
        loop over copyList for specific field
          if copyList % blockID == fieldIn_ptr % block % blockID
            copy data from fieldIn_ptr to fieldOut_ptr
          end if
        end loop
      end loop
    else
      pack data from fieldIn_ptr
      initiate mpi_isend for specific sendList
        end if
  end loop
end loop

loop over fieldOut list
  loop over recvList for specific field
    wait for mpi_irecv to finish
        unpack data into fieldOut_ptr
  end loop
end loop

loop over fieldIn list
  loop over sendList for specific field
    wait for mpi_isend to finish
  end loop
end loop
```

The only changes within the mpas_dmpar_exch_halo_field are internal, and only refer to the addition of checking copyList for shared memory copies. These changes should be similar to the internal changes to the allToAll routine changes presented within the pseudocode above.

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.

The previous interface for mpas_dmpar_get_owner_list can be seen below

```
subroutine mpas_dmpar_get_owner_list(dminfo,
                    nOwnedList, nNeededList,
                      ownedList, neededList,
```

```
                    sendList, recvList, inOffset)
```

where in this case, ownedList and neededList are simple arrays representing indices of owned and needed elements, nOwnedList and nNeededList are the number of elements in each respective list, sendList and recvList are output fields to represent the send and receive lists for the communications between these fields, and inOffset is an offset for receive lists.

The proposed new interface for this can be seen below

```
subroutine mpas_get_exchange_lists(dminfo, ownedListField,
                                   ownedDecomposed, neededListField,
                                      neededDecomposed, offSetField)
```

where ownedListField and neededListField are pointers to linked lists of 1d integer fields, ownedDecomposed and neededDecomposed are logical flags determining if the *ListField is decomposed using mpas_block_decomp or if there is one block per processor from the field, and offSetField represents a pointer to a linked list of 0d integer fields determining each blocks receive list offset.

The two major differences here are that the input data are given as fields rather than arrays, and the send/recv/copy lists are not output separate from the fields instead they are stored within the field structure. However these can be modified to put the send/recv/copy lists within the block rather than the field.

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.

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)
```

Because this routine relates specifically to cells, within the routine exchange lists are created for cells and stored within each block's cellsToSend, cellsToRecv, and cellsToCopy variables. This is done to keep in line with the previously identified requirement, and because the exchange lists that are actually used belong within this structure.

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 should be so they are isolated from the remainder of MPAS. In order to meet this requirement, a new module will be

created named mpas_block_creator.

The creation of this new module should allow a reorganization of the input and output routines. It should also allow the code writing for the creation of blocks to be more transparent to other MPAS developers.

One issue that comes up with the creation of this new module, is that MPI calls are now required within a module that's external to mpas_dmpar. Previously all MPI calls were isolated within the dmpar module, however because of some circular dependency issues that come up now this is no longer the case. Now, MPI calls can be restricted to the dmpar module and the block creation module.

Chapter 4

Testing

NOTE All of the testing described in this section relates only to the ocean core. Other core developers may test this with similar procedures but different simulations.

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 (1 block per proc).
- Finished branch simulation run with 8 processors and 8 blocks (1 block per proc).
- Finished branch simulation run with 1 processor and 8 blocks (8 blocks per proc).
- Finished branch simulation run with 2 processors and 8 blocks (4 blocks per proc).

If all of these simulations produce bit-for-bit output then testing can move on to a set of larger scale simulations.

- Current trunk 15km simulation with 1200 processors and 1200 blocks (1 block per proc).
- Finished branch simulation with 1200 processors and 1200 blocks (1 block per proc).
- Finished branch simulation with 600 processors and 1200 blocks (2 blocks per proc).
- Finished branch simulation with 24 processors and 1200 blocks (50 blocks per proc).

After these final four simulations show bit-for-bit output then the project can be deemed as completed.