Users Guide of Rocin and Rocout

X. Jiao and J. Norris

12/14/06

Part I
Rocin

1 Functionality

Rocin creates a series of Roccom windows by reading in a list of files, where the files can be in HDF or CGNS format. The files are self-contained in that they contain not only field data but also metadata (such as sizes, data types, locations, and units), which are needed to create Roccom windows. Rocin maps the "blocks" in the HDF files or "zones" in CGNS files into panes in Roccom windows. Rocin can be called collectively on multiple processes, or by a single process in a sequential program, in which MPI does not need to be initialized.

2 API

Rocin provides a simple API while maintaining the necessary flexibility and efficiency. It provides two sets of functions: the first set reads in metadata from the files (and can optionally read in array data as well), and the second set passes the array data from the files (or from Roccom's memory space if data are already loaded) to users memory space. The API are typically called through COM_call_function (see Roccom Users Guide) and hence all arguments are passed by pointers (references).

2.1 Read Window

To read metadata from files, a user can use one of the following two functions to read a single window or a series of windows, respectively.

- void read_window(const char *filename_patterns,
  const char *window_name,
  const MPI_Comm *comm=NULL,
  const RulesPtr *is_local=NULL,
  char *time_level=NULL,
  const char *str_maxlen=NULL);

- void read_windows(const char *filename_patterns,
  const char *window_prefix,
  const char *material_names=NULL,
  const MPI_Comm *comm=NULL,
  const RulesPtr *is_local=NULL,
  char *time_level=NULL,
  const char *str_maxlen=NULL);

The argument filename_patterns specifies a list of zero, one or more patterns (regular expressions) of the files to be read. Multiple patterns should be separated with empty spaces. If there is no file matching the patterns in filename_patterns on all processes (where filename_patterns can be empty or not), then a warning message will be printed, and an empty window (or windows) will be created. If there are matching files on any process, then a window (or windows) will be created by mapping from the matching files. It is guaranteed that each window defines all the attributes existing in any of the blocks (or zones) of its corresponding material in the files read by the processes within the given MPI communicator. If a pane has no data for a particular attribute in its corresponding block (or zone), then the array associated with the attribute will not be allocated in the pane.

The argument window_prefix specifies the window name (in the case of read_window()) or a prefix of the name(s) of the window(s) (for read_windows()) to be created. For read_windows, the argument material_names specifies a list of (space-separated) materials to be read from the files, and these strings are appended to window_prefix to obtain the complete window names. If the material_name is NULL or the empty string, then it is assumed that the files contain only one type of material, and the window name will be window_prefix. Note that the windows created by these functions must be deleted by the user by calling COM_delete_window after usage.

Among the remaining more advanced arguments, comm specifies an MPI communicator. If comm is not present or is NULL, then the default communicator is MPI_COMM_SELF. The arguments is_local is a function pointer of type

void (*)(const int &pid, const int &comm_rank, const int &comm_size, int *local),

which determines whether a pane of given block ID (HDF) or zone ID (CGNS) will be read by the current process. The first three arguments are input-only and the final argument is output-only. The last two arguments are for setting and obtaining the time level of the dataset. If time_level is a nonempty string, it will be used as an input, and the functions read only those datasets that have the matching time stamp; otherwise, all datasets are assumed to have the same time stamp, and that of the first dataset read from the files will be returned by copying up to *str_maxlen characters (including a null terminator) into time_level, if both time_level and str_maxlen are present and not NULL pointers.

2.2 Read by Control File

To allow more flexible user control, Rocin can also obtain HDF/CGNS file names and pane IDs from a user-provided control file.

- void read_by_control_file(const char *control_file_name,
  const char *window_name,
  const MPI_Comm *comm=NULL,
  char *time_level=NULL,
  const char *str_maxlen=NULL);

The control file contains a number of control blocks, each of which has up to four fields (a process rank, a list of file names, a list of pane IDs, and optionally, a material name), as described shortly, and each filed in general should be on one line. If present, the material name must be the same in all control blocks. At runtime, a process obtains the file names and pane IDs from the first control block that has a matching process rank. If comm is NULL, the default communicator is MPI_COMM_SELF, and the rank for all processes will be 0; if comm is present and is not MPI_COMM_NULL, then the process rank in the given MPI communicator will be used; if comm is MPI_COMM_NULL, then the rank of the current MPI process is replaced by a wild card, and the panes in all the files listed in all the control blocks will be read. In other words, when comm is MPI_COMM_NULL, all the panes in all the files listed in the control file will be considered local. This wild-card feature is useful for a serial application to read in all the panes, which would have been distributed by the control file onto different processes in a parallel run.

2.2.1 @Proc:

Marks the beginning of a process block, followed by a process's rank. A process reads in all the blocks that match the current process rank. A wild card '*' (without quotes) can be used after @Procs: to match any process. If the rank of the current process is a wild card (i.e., *comm==MPI_COMM_NULL), then all blocks will match the current process.

2.2.2 @Files:

A list of zero, one, or more file name patterns separated by empty spaces. A file name can contain the following place holders:

1. %dp for process rank, where d is an integer indicating the number of digits in the rank. If the number d is absent, then the default value is 4. If the current process's rank is a wild card (i.e., *comm==MPI_COMM_NULL), then any d digits in a file name will match.
2. %di for pane ID, where d is an integer indicating the number of digits in the pane ID. It maps a file with a pane ID n onto a process, if n will be mapped onto the current process by the pane mapping. The default value of d is 4.
3. %db for block ID, where d is an integer indicating the number of digits in the block ID. This option can only be used in conjunction with the @Block or @BlockCyclic mapping in the next subsection, which maps a file with a block ID n onto a process, if n*base will be mapped onto the current process by the mapping. The default value of d is 4.
4. %t for time stamp, which will be replaced by the time_level input argument.

For example, a file name "fluid*_%t_%4p.*" with a time level "00.000000" will be replaced by "fluid*_00.000000_0000.*" on process 0 and by "fluid*_00.000000_0001.*" on process 1. In general, a file name may use at most one of %dp, %di or %db. If the listed file names contain no directory path, then the files are assumed to be in the same directory that contains the control file. If a file name contains a relative path, then the path is considered to be relative to the current working directory at runtime.

2.2.3 @Panes:

Specifies a list of zero, one, or more pane IDs to be read by the process. For convenience, the user can also specify one of the following mapping rules:

- **@All** (or equivalently a wild card '*****' without quotes)

All panes are mapped onto the process.

- **@Cyclic [<offset>]**

A pane is mapped onto a process if

$$\text{mod}(\text{paneID} - \text{offset}, \text{nprocs}) = \text{rank}.$$

- **@BlockCyclic <base> [<offset>]**

A pane is mapped onto a process if

$$\text{mod}((\text{paneID-offset})/\text{base}, \text{nprocs}) = \text{rank}.$$

The default value of offset is 0. For example, for four processes, "@BlockCyclic 100 100" for 14 panes results in the following mapping:

Process 0: 100 500 900 1300
Process 1: 200 600 1000 1400
Process 2: 300 700 1100
Process 3: 400 800 1200

- **@Block <nblocks> <base> [<offset>]**

A pane is mapped onto a process if

$$\begin{cases} (\text{paneID} - \text{offset})/(\text{quot} * \text{base} + \text{base}) = \text{rank}, & \text{if rank} < \text{rem}, \\ (\text{paneID} - \text{offset} - \text{rem})/(\text{quot} * \text{base}) = \text{rank}, & \text{otherwise}, \end{cases}$$

where nblocks=quot*nprocs+rem. The default value of offset is 0. For example, for four processes, "@Block 14 100 100" results in the following mapping:

Process 0: 100 200 300 400
Process 1: 500 600 700 800
Process 2: 900 1000 1100
Process 3: 1200 1300 1400

Note that the @Panes field may be left out if the @Files field is empty. When *comm==MPI_COMM_NULL, then the @Panes field is immaterial.

2.2.4 @Material:

[To be implemented.] The keyword is followed by a character string to indicate the name of the material to be read. This field is optional and typically need not to be present when there is only one type of material in the files (i.e., when all the data in the files belong to the same window).

2.2.5 Sample Control Files

The following is a generic control file specifying each process to read in a rank-dependent file for a given time stamp, with block cyclic mapping for panes.

```
@Proc: *
@Files: fluid*_%t_%4p.hdf
@Panes: @BlockCyclic 100 1
```

The following is a specific control file for two processes.

```
@Proc: 0
@Files: fluid*_00.00_0000.hdf
@Panes: 1 3 5 7 9
@Material: fluid
@Proc: 1
@Files: fluid*_00.00_0001.hdf
@Panes: 2 4 6 8 10
@Material: fluid
```

2.3 Read Parameter File

To read parameters from a file into a window, the following function should be used:

- void read_parameter_file(const char *file_name,
  const char *window_name,
  const MPI_Comm *comm=NULL);

The function reads parameters from the given file and stores them as window attributes in the given parameter window. If the window already exists, then only the attributes that already exist in the window are read from the file. Otherwise a new window is created and all of the parameters are read in. Process 0 of the communicator should read the parameters, and then broadcast to all the processes. If comm is not specified, then the communicator of the window is used. If an option is listed more than once in the parameter file, the the last value for that option will overwrite the others.

2.4 Obtain Attribute

To obtain array data from files, the following function should be used:

- void obtain_attribute(const Attribute *attribute_in,
  Attribute *attribute_user,
  int *pane_id=NULL);

This function fills the second (destination) attribute from the files using the data corresponding to the first (source) attribute. The destination and source attributes can be the same. The attributes could be a user-defined attribute, or an aggregate attribute, such as "window.conn", "window.mesh", "window.pmesh", "window.atts", and "window.all", which indicate obtaining connectivity tables only, mesh only (nodal coordinates and connectivity tables), mesh with pane connectivity, attributes (everything except for pmesh), and everything (including pmesh and attributes), respectively. If the third argument is present and is nonzero, then only the pane with the given ID will be copied.

2.5 Initialization and finalization

Rocin provides the following routines for initialization and finalization.

- extern "C" void Rocin_load_module(const char *module_name);

Usually this procedure is invoked by COM_load_module("Rocin", module_name). It creates a window with name <module_name> in Roccom and register its functions into the window.

- extern "C" void Rocin_unload_module(const char *module_name);

This procedure is typically invoked by COM_unload_module("Rocin", module_name). It unloads the module from Roccom by deleting the window created by Rocin_load_module.

3 Implementation notes

In read_window, in general, only metadata are read into memory to create windows. The data buffers of the windows may or may not be allocated yet. In obtain_attribute, Rocin obtains data from the files to fill in user buffers. However, for certain file formats, an implementation of Rocin may read in physical data during read_window as well. The downside of the latter approach is higher memory requirements.

The function obtain_attribute can permute memory layout of an attribute. In general, an attribute in Rocin can have either staggered or contiguous layout with a stride 1, but the user attribute can have either staggered or contiguous layout and can also have a stride other than 1. The function obtain_attribute support all these layouts.

The functions in the API can be implemented as C++ static member functions of Rocin, or regular member functions. In the former case, the functions are registered with Roccom using COM_set_function; in the latter case, they are registered using COM_set_member_function. Rocin works even if MPI_Init was not called. Rocin must be Charm-safe in the sense that there is no global (or static) variable [Current implementation is not yet Charm-safe].

Part II
Rocout

4 Functionality

Rocout writes a given attribute in a Roccom window into a file in one of the supported formats (HDF and CGNS), which can be read by application codes through Rocin, and by Rocketeer (and CGNS-compliant tools, such as Tecplot, for CGNS format) for visualization. Rocout can support background output by creating an I/O thread to allow overlap computation with I/O. The interface of Rocout is consistent with Rocpanda, the parallel I/O utility.

5 API

Similar to Rocin, Rocout API typically should be called through COM_call_function.

5.1 Output

- void write_attribute(const char *filename_pre,
  const COM::Attribute *attr,
  const char *material,
  const char *timelevel,
  const char *mfile pre = NULL,
  const MPI_Comm *comm=NULL,
  const int *pand_id=NULL);

This function writes an attribute of local panes or of the pane with the given Pane ID (*pane_id, if present) into a file, where the file name is <fname_pre><process_rank>.<suffix>, where <process_rank> is the rank of the given MPI process, whose number of digits can be controlled by set_option() (see below). This function will either overwrite the file if the output mode (set by set_option()) is "w" or append to the file if the mode is "a".

If mfile_pre is not null and nonempty, then the output file will make a reference to the file <mesh_pre><process_rank>.<s
for the pmesh data with the same material name, and write only non-pmesh data into the current file. When appending data attributes into a file that already contains the pmesh, then mfile_pre should be the same as filename_pre.

When calling write_attribute multiple times to write several datasets into the same set of HDF files, it is important that the write operations for different panes must not interleave (i.e., the data for the same pane must be written out consecutively). In general, different windows can be written into the same set of files, but these windows must have different material names.

Parameters:

1. fname_pre: the prefix of the file name, which can contain the directory part of the file.
2. attr: a reference to the attribute to be written. The given attribute can be either a user defined attribute, or one of the following aggregate attributes: "window.mesh" (coordinates and connectivity), "window.pmesh" (mesh with pane connectivity), "window.atts" (all the data in the pane except for pmesh), or "window.all" (all the data).
3. material: the material name to distinguish different windows. It is recommended that different windows use different material names, and is required if more than one window is written into the same HDF/CGNS file.
4. timelevel: a time stamp of the dataset.

5. mfile_pre: the prefix of the name of the file that contains the pmesh data of the given attribute. If not present or is empty, then the pmesh will be written along with the given data attributes. If mfile_pre does not start with "/" (i.e., does not have an absolute path), then the path of the mesh file must be either relative to the directory for fname_pre (with higher precedence) or relative to the current working directory (with lower precedence).

6. comm: the MPI communicator in which the process rank should be obtained. If comm is NULL, then the default value is the communicator of the owner window of the attribute (note that the default value is different from that with Rocin::read_window).

7. pane_id specifies the pane (or panes) to be written. If pane_id=NULL or *pane_id=0, then all panes will be written. If *pane_id>0, then only that specific pane will be written. It is an error if *pane_id<0.

Instead of using set_option to control the output mode, a user can also use one of the following two functions, which correspond to overwrite and append, respectively. [To be implemented.]

- void put_attribute(const char *filename_pre,
  const COM::Attribute *attr,
  const char *material,
  const char *timelevel,
  const char *mfile pre = NULL,
  const MPI_Comm *comm=NULL,
  const int *pand_id=NULL);

- void add_attribute(const char *filename_pre,
  const COM::Attribute *attr,
  const char *material,
  const char *timelevel,
  const char *mfile pre = NULL,
  const MPI_Comm *comm=NULL,
  const int *pand_id=NULL);

These functions take the same arguments as write_attribute.

5.2 Metadata Output

- void write_rocin_control_file(const char *window_name,
  const char *file_prefixes,
  const char *control_file_name);

This function generates a control file for Rocin for the given window and datafile prefixes. This control file can be used with Rocin's read_by_control_file member function.

- void write_parameter_file(const char *file_name,
  const char *window_name,
  const MPI_Comm *comm=NULL);

This function writes out the parameters defined in the given window to a parameter file. Only process 0 of the MPI communicator writes the file. If comm is NULL, then the communicator of the window associated with window_name is used.

5.3 Synchronization

- void sync();

Wait for the completion of an asynchronous write operation. It is needed only if the "async" mode is set to "on" by set_option, described as follows.

5.4 Control

- void set_option(const char *option_name,
  const char *option_val);

Set an option for Rocout, such as controlling the output format. The currently supported option_name and their potential values are:

"format": with values "HDF" and "CGNS" (default is "HDF").
"async": with values "on" and "off" for enabling/disabling background out, respectively (default is off).
"mode": with values "w" and "a" (corresponding to overwrite the file and append to the file), which control the output mode of write_attribute (default is "w").
"localdir": a directory path to prepend to the filename prefixes given to write_attribute, put_attribute and add_attribute (default is "").
"rankwidth": the width of the process-rank to be appended to the filename_pre and mesh_pre (default is "4"). If zero, then do not append process rank.
"pnidwidth": the width of the pane ID to be appended to the filename_pre and mesh_pre after appending process rank. Default value is 0, for which the pane ID is not appended.
"separator": the character to use to separate the rank and pane ids in generated filenames (default is "_"). A separator is only used if both "rankwidth" and "pnidwidth" are non-zero.
"errorhandle": with values "abort", "ignore", or "warn".
"ghosthandle": with values "write" and "ignore".

Option names and values are case-sensitive.

- void read_control_file(const char *filename);

This function allows the user to set Rocout options by means of a control file. The given file should be a list of option name/values pairs, separated by an equals sign. For example:

format = CGNS
localdir = /turing/projects/csar/MyDataDir
errorhandle = abort

Any option name supported by set_option may be used.

5.5 Initialization and finalization

As Rocin, Rocout provides the following routines for initialization and finalization.

- extern "C" void Rocout_load_module(const char *module_name);

Usually this procedure is invoked by COM_load_module("Rocout", module_name). It creates a window with name <module_name> in Roccom and register its functions into the window.

- extern "C" void Rocout_unload_module(const char *module_name);

This procedure is typically invoked by COM_unload_module("Rocout", module_name). It unloads the module from Roccom by deleting the window created by Rocout_load_module.

6 Implementation notes

The functions in the API can be implemented as C++ static member functions or regular member functions of Rocout. In the former case, the functions are registered with Roccom using COM_set_function; in the latter case, they are registered using COM_set_member_function. Rocout works even if MPI_Init was not called, in which case the rank is assumed to be 0. Rocout must be Charm-safe in the sense that there is no global (or static) variable.

A Sample Code

Samples codes of Rocin and Rocout can be found under Roccom/Rocin/test and Roccom/Rocout/test, respectively.

B Contributors

John Norris is the main developer of Rocin and Rocout. Phil Alexander helped with debugging Roccom calls in Rocin and Rocout. Jim Jiao is responsible for the specification, with participation of Mike Campbell in the early stage and many helpful discussions with Andreas Haselbacher and Orion Lawlor. Jim has also been indispensible in debugging Rocin and Rocout. Orion wrote a sample code printin.C for Rocin.