

Rocom Users Guide

Version 3.0

Xiangmin Jiao, Gengbin Zheng

Revised: 7/20/2008

1 Introduction

Large-scale numerical simulation of a complex system, such as a solid rocket motor, requires consideration of multiple physical components, such as fluid dynamics, solid mechanics, and combustion. Because of this multidisciplinary nature, development of such a simulation code typically involves collaboration among many research subgroups using a partitioned approach, in which the individual physics codes are developed more or less independently of the integration effort. The objective of Rocom is to ease the integration of such independently developed modules into a coherent, coupled system, particularly in a distributed parallel setting. It is designed to maximize concurrency in development of different modules, minimize user effort in software integration, and provide interoperability between different programming languages (in particular, C, C++, and Fortran 90).

The motivating multiphysics application for the integration framework described in this paper is an ongoing project at the Center for Simulation of Advanced Rockets (CSAR) at the University of Illinois (<http://www.csar.uiuc.edu>). The ultimate objective of CSAR is to develop an integrated software system for detailed whole-system simulation of solid rocket motors under normal and abnormal operating conditions. Figure 1 shows the overall structure of the current generation of our simulation code. Rocom categorizes

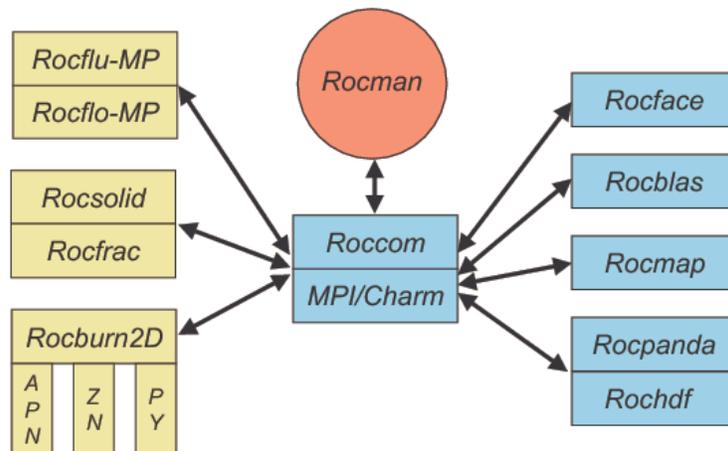


Figure 1: Software components of Rocstar.

these modules into two types: *application modules* (including *computation (physics) modules* and *orchestration modules*) and *framework modules* (including Rocom's runtime system and the *service modules*). In the

figure, the boxes on the left show the physics modules. On the top is the orchestration module, *Rocman* (and its modern version *Rocman3*), which manages the coupling algorithms. On the right are the computer science modules that provide services to the physics and orchestration modules through *Rocom*. The physics modules are written in Fortran 90, and the service modules are written in C++. The parallel implementation uses the standard Message Passing Interface (MPI) for all modules except *Rocflu-MP*, which uses the finite volume framework of Charm++.

Although it was motivated specifically by the needs of the rocket simulation application just described, the integration framework we have developed is quite general, and should be equally applicable to many other multiphysics simulations involving multiple, interacting software modules representing various physical components, especially those based on spatial decomposition into geometric domains with associated meshes. *Rocom* provides systematic methods for modules in a complex simulation to keep track of their data and to access data defined by other modules. Besides declaring variables and allocating buffers, each computation module registers its datasets with *Rocom*. These datasets can later be retrieved from *Rocom* by the same module or other modules, using parameters such as data block number, attribute name, etc. Functions can be registered and invoked in a similar way through *Rocom*. This scheme allows great independence in design and development of individual modules, hides the coding details of different research subgroups, and enables plug-and-play of different modules in Rocstar.

Rocom is composed of three parts: a simple API (Application Programming Interface) for application modules, a C++ interface for developing service modules, and a runtime system. The API provides subroutines for registering the public data and functions of a module, querying a publicized data of a module, and invoking registered functions. In general, the API is the only part that application code developers need to learn in order to use *Rocom*. After an application code registers its data with *Rocom*, it can easily take advantage of the service utilities built on top of *Rocom*'s developers interface (such as parallel I/O). *Rocom* also provides support for eliminating global variables from application codes, which can then take advantage of the multi-thread and dynamic load balancing features provided by Charm++. In this documentation, we address only the general concepts of *Rocom* and its API. For a more in-depth discussion on the developers interface or runtime system, please see the *Rocom* Developers Guide.

1.1 Historical Notes and Acknowledgments

The design of *Rocom* was first proposed by Xiangmin Jiao and Michael Campbell in December 2000, with the participation of Jie Zheng (Prof. Eric de Sturler's former student) and Milind Bhandarkar. The proposed interface and underlying concepts were described in Version 1.0 of the document, which was initially called the "Window Manager Users guide" and was written before *Rocom* was implemented. Nevertheless, Version 1.0 served as a design specification of a bulk of *Rocom*.

The name *Rocom* was proposed by Prof. Michael T. Heath, and so was the term "window". The term "pane" was due to Heath and Bill Dick. These terms were inspired by the observation that the interface data are frequently lower-dimensional objects of a domain and are the "window" through which a module is communicating with the outside world.

Rocom Version 1.0 was implemented by Jiao in 2001. Version 2.0 has been rewritten substantially since Version 1.0, with some new features introduced by Jiao as described in the previous subsection. These changes affected almost all interface subroutines. Version 2.x was implemented in Fall 2002 by Jiao. Version 3.0 was motivated by the needs of dynamic arrays, for Rocin, and for mesh adaptivity. Thanks to Andreas Haselbacher's input on improving the presentation of this document.

1.1.1 New Changes in Version 3.x

The following features have been recently added into Roccom.

1. Added support for dynamic arrays by differencing size and capacity of an array.
2. Simplified and unified interface for initialization of mesh and other attributes using `set_size` and `set_array`.
3. Specification of data layout during `set_array`.
4. Ability to redefine attributes (such as “nc”) and to define new connectivity types (to be implemented).
5. Differentiates `allocate_array` and `resize_array` (conditional reallocation).
6. More comprehensive support for inquiring information with `COM_get_panes`, `COM_get_attributes`, and `COM_get_connectivities`. .
7. Better support for mixed meshes.
8. New keywords “pconn”, “pmesh”, and “atts” and new name format for element connectivity.
9. Added support for setting, obtaining, and checking bounds of data attributes.
10. Added support for registration of C++ member function function.
11. Added `set_array_const` and `get_array_const` for protecting data integrity through immutable data.
12. Added C++ version of Roccom APIs which support `std::string` as parameters.

1.1.2 Changes in Version 2.x

The following features were added into Roccom.

1. Inheritance of windows.
2. Profiling tools.
3. Concept of member function.
4. Attribute and function handles.
5. Support for Fortran 90 pointers.
6. Loading and unloading of dynamic libraries.
7. Memory management. (Version 2.1 allows allocation and deallocation of attributes for individual panes).
8. More element types (pyramids and wedges/prisms added in Version 2.1).

2 Overview

Rocom (standing for Rocstar Component Object Manager) is a component-based, object-oriented, data-centric software integration framework, which provides a systematic, object-oriented, data-centric approach for inter-module interaction. Under this framework, a computation module constructs distributed objects called *windows* and registers its datasets into windows. With the authorization of their owner modules or the orchestration module, these datasets can later be retrieved from *Rocom* by other modules using handles provided by *Rocom*. Functions can be registered and invoked similarly through *Rocom*. This scheme allows great independence in design and development of individual modules, hides the coding details of different research subgroups, and provides additional features such as automatic tracing and profiling.

2.1 Object-Oriented Interfaces

To simplify inter-module interfaces, *Rocom* utilizes an object-oriented methodology for abstracting and managing the data and functions of a module. This abstraction is mesh- and physics-aware and supports encapsulation, polymorphism, and inheritance.

2.1.1 Windows and Panes

Rocom organizes data and functions into distributed objects called *windows*. A window encapsulates a number of *data attributes* (such as the mesh and some associated field variables) and public *functions* of a module, any of which can be empty. A window can be partitioned into multiple *panes*, for exploiting parallelism or for distinguishing different material or boundary-condition types. In a parallel setting, a pane belongs to a single process, while a process may own any number of panes. All panes of a window must have the same types of data members, although the sizes of data members may vary. A module constructs windows at runtime by creating attributes and registering the addresses of the attributes and functions. Typically, the attributes registered with *Rocom* are *persistent* (instead of temporary) datasets, in the sense that they live throughout the simulation (except that windows may need to be reinitialized at some events, such as remeshing). Different modules can communicate with each other only through windows, as illustrated in Figure 2.

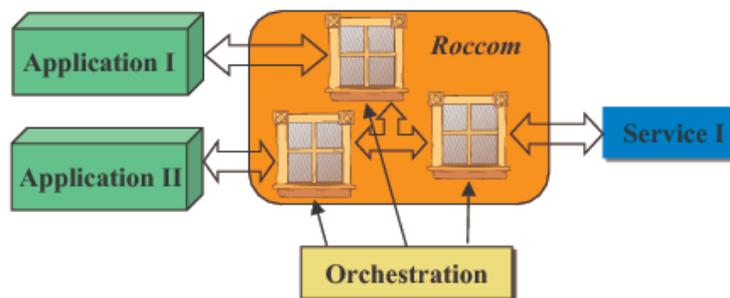


Figure 2: Schematic illustration of windows and panes.

A code module references windows, attributes, or functions using their names, which are of character-string type. Window names must be unique across all modules, and an attribute or function name must be unique within a window. A code module can obtain an integer *handle* of (i.e., a reference to) an attribute/function from *Rocom* with the combination of the window and attribute/function names. The handle of an attribute

can be either *mutable* or *immutable*, where an immutable handle allows only read operations to its referenced attribute, similar to a const reference in C++. Each pane has a user-defined positive integer ID, which must be unique within the window across all processors but need not be consecutive.

2.1.2 Data Attributes

Data attributes of a window include mesh data, field variables, and window or pane attributes. The former two types of attributes are associated with nodes or elements. A nodal or elemental attribute of a pane is conceptually a two-dimensional dataset: one dimension corresponds to the nodes/elements, and the other dimension corresponds to the data within a node/element. The dataset can be stored in a row- or column-major two-dimensional array, or be stored in separate arrays for each component of the dataset. *Rocom* allows users to specify a stride (the distance in the base data type, such as int or double precision) between the same component of two consecutive items (such as nodes/elements).

Mesh Data Mesh data include nodal coordinates, pane connectivity, and element connectivity (or simply connectivity), whose attribute names and data types are predefined by *Rocom*. The nodal coordinates are double-precision floating-point numbers, with three components per node. The pane connectivity specifies the communication patterns between nodes shared by two or more panes, and is a pane attribute packed in a 1-D array. The registration of pane connectivity is desirable for many purposes, but it is optional and can be computed automatically from coordinates using Rocmap.

Rocom supports both surface and volume meshes, which can be either multi-block structured or unstructured with mixed elements. For multi-block meshes, each block corresponds to a pane in a window. For unstructured meshes, each pane has one or more connectivity tables, where each connectivity table contains consecutively numbered elements (i.e., their corresponding field variables are stored consecutively) of the same type. Each connectivity table must be stored in an array with contiguous or staggered layout. To facilitate parallel simulations, *Rocom* also allows a user to specify the number of layers of ghost nodes and cells for structured meshes, and the numbers of ghost nodes and cells for unstructured meshes.

Field Variables Field variables are nodal or elemental attributes that have no designated names or data types. A user must first define such an attribute in the window and then register the addresses of the attribute for each pane. For a specific pane, if a field variable is stored in one single array, then the array is registered with a single call; if it is stored in multiple arrays, then the user must register these arrays separately.

Window and Pane Attributes A data member can also be associated with either the window or a pane. Examples of window attributes include a data structure that encapsulates the internal states of a module or some control parameters. An example of a pane attribute is an integer flag for the boundary condition type of a surface patch. Similar to field variables, these attributes do not have designated names or data types, and must be created within a window and then registered.

Aggregate Attributes In *Rocom*, although attributes are registered as individual arrays, attributes can be referenced as an aggregate. For example, the name “mesh” refers to the collection of nodal coordinates and element connectivity; the name “all” refers to all the data attributes in a window. For staggered attributes, one can use “*i*-attribute” ($i \geq 1$) to refer to the *i*th component of the attribute or use “attribute” to refer to all components collectively. See Section 3.2.2 for details.

Aggregate attributes enable high-level inter-module interfaces. For example, one can pass the “all” attribute of a window to a parallel I/O routine to write all of the contents of a window into an output file with a single call. As another example, it is sometimes more convenient for users to have *Roccom* allocate memory for data attributes and have application codes retrieve memory addresses from *Roccom*. *Roccom* provides a call for memory allocation, which takes a window attribute name pair as input. A user can pass in “all” for the attribute name, which will have *Roccom* allocate memory for all the unregistered attributes.

2.1.3 Functions

A window can contain not only data members but also function members. A module can register a function into a window, to allow other modules to invoke the function through *Roccom*. Registration of functions enables a limited degree of runtime polymorphism. It also overcomes the technical difficulty of linking object files compiled from different languages, where the mangled function names can be platform and compiler dependent.

Member Functions Except for very simple functions, a typical function needs to operate with certain internal states. In object-oriented programs, such states are encapsulated in an “object”, which is passed to a function as an argument instead of being scattered into global variables as in traditional programs. In some modern programming language, this object is passed implicitly by the compiler to allow cleaner interfaces.

In mixed-language programs, even if a function and its context object are written in the same programming language, it is difficult to invoke such functions across languages, because C++ objects and F90 structures are incompatible. To address this problem, we introduce the concept of member functions of attributes into *Roccom*. Specifically, during registration a function can be specified as the member function of a particular data attribute in a window. *Roccom* keeps track of the data attribute and passes it implicitly to the function during invocation, in a way similar to C++ member functions. Because the caller no longer needs to know the context object of the callee, this concept overcomes the incompatibility without sacrificing object-orientedness.

Optional Arguments *Roccom* supports the semantics of optional arguments similar to that of C++ to allow cleaner codes. Specifically, during function registration a user can specify the last few arguments as optional. *Roccom* passes null pointers for those optional arguments whose corresponding actual parameters are missing during invocation.

2.1.4 Inheritance

In multiphysics simulations, inheritance of interface data is useful in many situations. First, the orchestration module sometimes needs to create data buffers associated with a computation module for the manipulation of jump conditions. Inheritance of windows allows the orchestration module to obtain a new window for extension or alteration, without altering the existing window. Second, a module may need to operate on a subset of the mesh of another module. In rocket simulation, for example, the combustion module needs to operate on the burning surface between the fluid and solid. Furthermore, the orchestration module sometimes needs to split a user-defined window into separate windows based on boundary-condition types, so that these subwindows can be treated differently (e.g., written into separate files for visualization). Figure 3 depicts a scenario of inheritance among three windows.

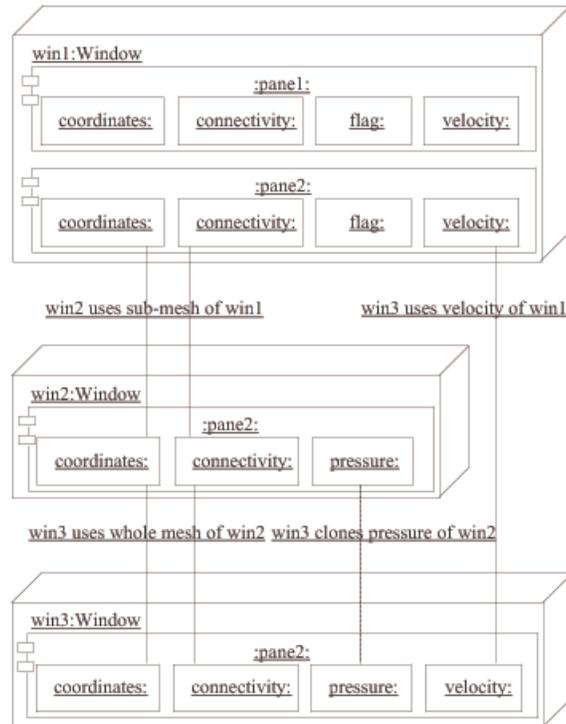


Figure 3: Scenario of inheritance of mesh and field attributes among three windows.

To support these needs, *Rocom* allows inheriting the mesh from a parent window to a child window in either of two modes. First, the mesh can be inherited as a whole. Second, only a subset of panes that satisfy a certain criterion are inherited. After inheriting mesh data, a child window can inherit data members from its parent window, or other windows that have the same mesh (this allows for *multiple inheritance*). The child window obtains the data only in the panes it owns and ignores other panes. During inheritance, if an attribute already exists in a child window, *Rocom* overwrites the existing attribute with the new attribute.

Rocom supports two types of inheritance for data members: cloning (with duplication) and using (without duplication). The former allocates new memory space and makes a copy of the data attribute in the new window, and is safer in terms of data integrity. The latter makes a copy of the references of the data member, which avoids the copying overhead associated with cloning and guarantees data coherence between the parent and child, and is particularly useful for implementing orchestration modules.

2.1.5 Data Integrity

In complex systems, data integrity has profound significance for software quality. Two potential issues can endanger data integrity: dangling references and side effects. *Rocom* addresses these issues through the mechanisms of persistency and immutable references, respectively.

Persistency *Rocom* maintains references to the datasets registered with its windows. To avoid dangling references associated with data registration, *Rocom* imposes the following persistency requirement: the datasets registered with a window must outlive the life of the window. Under this model, any persistent object can refer to other persistent objects without the risk of dangling references. In a heterogeneous

programming environment without garbage collection, persistency cannot be enforced easily by the runtime systems instead, it is considered as a design pattern that application code developers must follow.

Immutable References Another potential issue for data integrity is side effects due to inadvertent changes to datasets. For the internal states of the modules, *Roccom* facilitates the traditional integrity model through member functions described earlier. In *Roccom*, a service module can obtain accesses to another module's data attributes only through its function arguments, and *Roccom* enforces at runtime that an immutable handle cannot be passed to mutable arguments.

2.2 Architecture of *Roccom*

The core of *Roccom* is composed of three parts: an Application Programming Interface (API), a C++ class interface for development of service modules, and a runtime system for the bookkeeping associated with data objects and invocation of functions.

2.2.1 *Roccom* API

The *Roccom* API supplies a set of primitive function interfaces to physics and orchestration modules for system setup, window management, information retrieval, and function invocation. The subset of the API for window management serves essentially the same purpose as the Interface Definition Language (IDL) of other frameworks (such as CCA), except that *Roccom* parses the definitions of the windows at runtime. *Roccom* provides different bindings for C++ and F90, with similar semantics. See Section 3 for details.

2.2.2 C++ Class Interfaces

Roccom provides a unified view of the organization of distributed data objects for service modules through the abstractions of windows and panes. Internally, *Roccom* organizes windows, panes, attributes, functions, and connectivities into C++ objects, whose associations are illustrated in Figure 4, on a UML class diagram. A Window object maintains a list of its local panes, attributes, and functions; a Pane object contains a list of attributes and connectivities; an Attribute object contains a reference to its owner window. By taking references to attributes as arguments, a function can follow the links to access the data attributes in all local panes. The C++ interfaces conform to the principle of deeply immutable references, ensuring that a client can navigate through only immutable references if the root reference was immutable. Through this abstraction, the developers can implement service utilities independently of application codes, and ensure applicability in a heterogeneous environment with mixed meshes, transparently to physics modules.

2.2.3 *Roccom* Runtime System

The runtime serves as the middleware between modules. It keeps track of the user-registered data and functions. During function invocation, it translates the function and attribute handles into their corresponding references with an efficient table lookup, enforces access protection of the attributes, and checks whether the number of arguments of the caller matches the declaration of the callee. Furthermore, the runtime system also serves as the middleware for transparent language interoperability. For example, if the caller is in F90 whereas the callee is in C++, the runtime system will null-terminate the character strings in the arguments before passing to the callee.

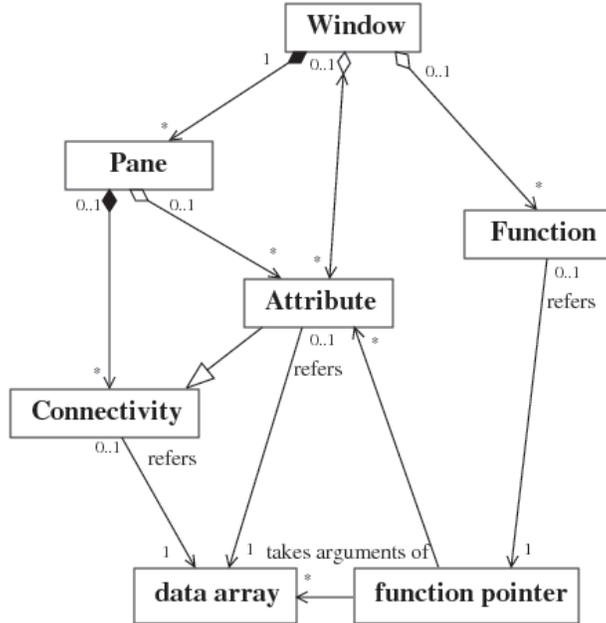


Figure 4: UML associations of *Rocom*'s classes.

Through the calling mechanism, *Rocom* also provides tracing and profiling capabilities for inter-module calls to aid in debugging and performance tuning. It also exploits hardware counters through PAPI to obtain performance data such as the number of floating-point instructions executed by modules. A user can enable such features using command-line options without additional coding. To further ease debugging, we have integrated the malloc() debugger Electric Fence (<http://perens.com/FreeSoftware>) into the framework. Electric Fence stops a program on the first instruction that causes a bounds violation, and requires no recompilation but only relinking of the executable. Together with the tracing capability, this tool makes it much easier to identify which module is responsible for a memory violation.

2.3 Module Requirements

A *Rocom* application has a driver or orchestration module, in this case *Rocman*, which is responsible for system setup and invoking the registered functions in turn. Each *Rocom*-compliant module must provide a load-module routine, which creates a window to encapsulate its interface functions and context objects, and an unload-module, which destroys the window, where the window name is typically the same as that of the module. By calling the load-module routines, the driver dynamically loads a set of modules into the runtime system. Through *Rocom*'s calling mechanism, *Rocman* then invokes the functions of the physics and service modules, which in turn can also invoke functions provided by other modules.

Furthermore, a physics module must provide three basic interface functions: an initialization routine for allocating and initializing internal data structures and creating two windows (one for jump conditions and one for internal data to be stored/restored for restart and predictor-corrector iterations); a main update routine for updating the physical solution for one system time step by obtaining jump conditions from the orchestration module; a finalization routine for deallocating memory and destroying the windows. Figure 5 shows a schematic UML sequence diagram of the interactions between the modules of Rocstar.

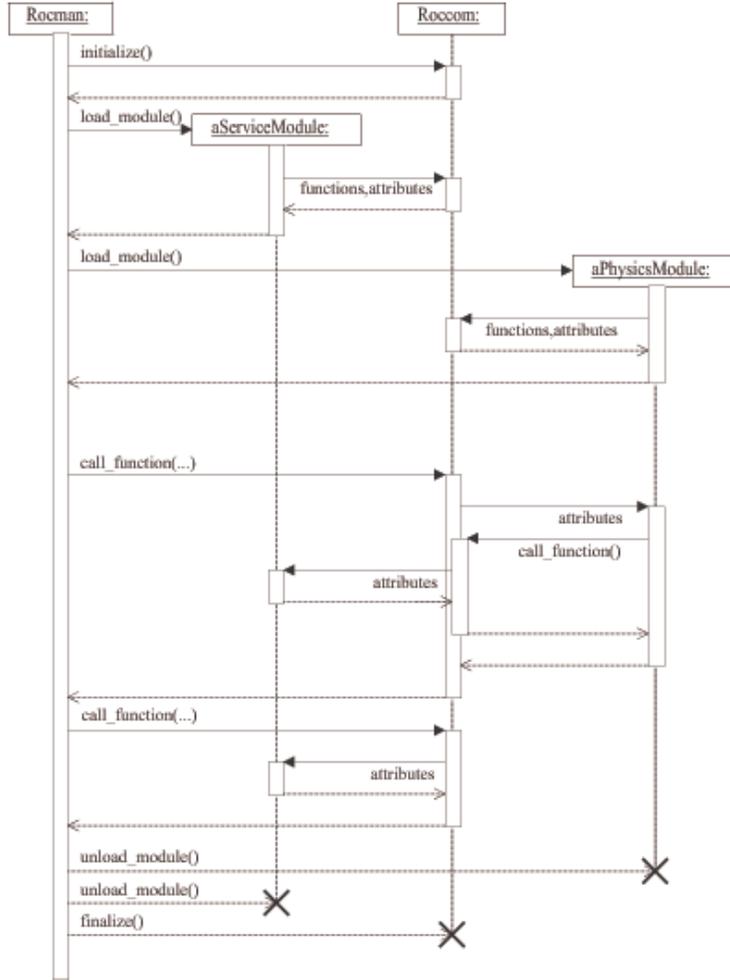


Figure 5: Interaction between modules shown on UML sequence diagram.

3 Roccom API

Roccom provides different bindings for C, C++, and Fortran 90, with similar semantics, except that C/C++ is case-sensitive whereas Fortran is case-insensitive, and C/C++ passes arguments by value whereas Fortran passes by reference. Another subtle difference is that Fortran character strings, which are not null terminated by default, must be interpreted differently from C/C++ character strings. These differences are apparent in the prototype definitions of the subroutines, but are mostly transparent to users. Roccom's interface prototypes are defined in "roccom.h" (for C/C++) and "roccomf90.h" (for Fortran 90), which must be included by the codes in corresponding languages, respectively.

Roccom's interface subroutines follow the following conventions: They all start with the prefix `COM_`, followed by lower-case letters (for C/C++). Most subroutines return no values unless otherwise specified. If a non-fatal error occurred inside a Roccom subroutine, an error flag will be set. For the C and Fortran interfaces, the error code can be obtained by calling `COM_get_error_code`. For the C++ interface, the error code will be thrown as an exception. This error-handling convention is different from Roccom 1.0, where every subroutine returns either zero or a nonzero error code.

Although Roccom's API has about 40 functions, a simple computation module needs to use only about 10 of

them, mostly in Section 3.2. The other functions are more advanced and meant for the orchestration module and for more complex physics modules.

3.1 Initialization and Finalization

3.1.1 Startup and Shutdown of Roccom

Implementations of Roccom’s runtime system require some setup operations before any other Roccom operations can be performed. To provide for this, Roccom includes an initialization subroutine `COM_init`.

```
C:      COM_init(int *argc, char ***argv)
Fortran: SUBROUTINE COM_INIT
```

This subroutine must be called exactly once from every process before any other Roccom subroutine (apart from `COM_initialized`) is called. It is typically called from the driver routine of the application. The C version accepts the arguments `argc` and `argv`, which are the arguments of the main routine of C. `COM_init` parses the following options:

- “-com-v *n*”: Set the verbose level of all processes to *n* (see `COM_set_verbose`);
- “-com-vp *n*”: Set the verbose level of process *p* to *n* (see `COM_set_verbose`);
- “-com-mpi”: Call `MPI_Init` within `COM_init`.
- “-com-home <directory>”: Search for shared libraries under <directory>/lib. Alternatively, one can pass the directory by setting either `ROCCOM_HOME` or `ROCSTAR_HOME` environment variables.

The Fortran version `COM_INIT` takes no arguments. A corresponding subroutine `COM_finalize` is also provided for Roccom to clean up its state after the execution of the program. It also needs to be called on every process. Once this subroutine is called, no Roccom subroutine may be called.

```
C:      COM_finalize(void)
Fortran: SUBROUTINE COM_FINALIZE
```

The following is a piece of code in C that illustrates its usage.

```
int main(int argc, char **argv) {
    COM_init(&argc, &argv);
    /* main program */
    COM_finalize();
}
```

Roccom provides a subroutine `COM_initialized` for checking whether `COM_init` has been called.

```

C:      int COM_initialized(void)
Fortran: FUNCTION COM_INITIALIZED()
        INTEGER :: COM_INITIALIZED

```

The argument flag is set to nonzero if `COM_init` has been called and zero otherwise.

Furthermore, Roccom runtime environment can also be shut down abnormally by calling `COM_abort`.

```

C:      void COM_abort(int ierr)
Fortran: SUBROUTINE COM_ABORT(IERR)
        INTEGER, INTENT(IN) :: IERR

```

This function terminates a Roccom program and returns the error code `ierr` to the invoking environment. If MPI was initialized, then `COM_abort` calls `MPI_Abort` internally on `MPI_COMM_WORLD`. Otherwise, it calls the `exit` function of the standard C library. For Fortran codes, Roccom also provides a related subroutine `COM_CALL_EXIT`, which we describe in Section 3.3.6.

3.1.2 Loading and Unloading of Modules

In the Roccom framework, each module can be built into a dynamic library named `libRocfoo.so`. Roccom provides the following interface to load and unload the dynamic library for a module.

```

C:      COM_load_module(const char *modName, const char *winName)
Fortran: SUBROUTINE COM_LOAD_MODULE(MODNAME, WINNAME)
        CHARACTER(*), INTENT(IN) :: MODNAME, WINNAME

C:      COM_unload_module(const char *modName, const char *winName=NULL)
Fortran: SUBROUTINE COM_UNLOAD_MODULE(MODNAME,WINNAME)
        CHARACTER(*), INTENT(IN) :: MODNAME,WINNAME
        OPTIONAL :: WINNAME

```

The argument `modName` is the main part of the library name (e.g., “Rocfoo” for `libRocfoo.so`). The module `Rocfoo` needs to supply two subroutines, `Rocfoo_load_module` and `Rocfoo_unload_module`, which takes `winName` as its argument. When `COM_load_module/COM_unload_module` is called, Roccom locates the symbol `Rocfoo_load_module/Rocfoo_unload_module` in `libRocfoo.so`, respectively, and invokes these user-provided routines by passing `winName` to load/unload the module. For `COM_unload_module`, the `winName` argument can be omitted if the module is loaded only once.

For ease of debugging, sometimes it is desirable to build Roccom modules and applications statically. In this case, an application need to define the prototypes of `Rocfoo_load_module` and `Rocfoo_unload_module` and call them directly instead of through `COM_load_module/COM_unload_module`. Roccom provides the following macros to C/C++ codes (defined when `roccom.h` is included):

```

COM_EXTERN_MODULE( modName_noquotes)
COM_LOAD_MODULE_STATIC_DYNAMIC( modName_noquotes, winName)
COM_UNLOAD_MODULE_STATIC_DYNAMIC( modName_noquotes, winName)

```

Depending on whether the macro `STATIC_LINK` is defined (e.g., by passing `-DSTATIC_LINK` to the compiler) or not, these macros expands to different statements. When `STATIC_LINK` is defined, then `COM_EXTERN_MODULE(Rocfoo)` defines `Rocfoo_load_module` and `Rocfoo_unload_module`. Otherwise, it expands to `noop`. For C++ codes, since `COM_EXTERN_MODULE` uses the extern “C” modifier, it cannot be used inside a function. `COM_LOAD_MODULE_STATIC_DYNAMIC(Rocfoo, “FOO”)` expands to `Rocfoo_load_module(“FOO”)` if the `STATIC_LINK` is defined, and to `COM_load_module(“Rocfoo”, “FOO”)`, otherwise; similarly for `COM_UNLOAD_MODULE_STATIC_DYNAMIC`.

3.2 Data and Function Registration

Rocom organizes data and functions into windows. A window encapsulates a number of data members (such as the mesh and some associated data attributes) and public functions of a module. A module can create any number of windows. All panes of a window must have the same collection of data members, although the size of each data member may vary.

3.2.1 Creation of Window

A call to `COM_new_window` creates an empty window with a given name.

```
C++:    COM_new_window(const std::string wName, MPI_Comm comm=MPI_COMM_SELF)
C:      COM_new_window(const char *wName, MPI_Comm comm=MPI_COMM_SELF)
Fortran: SUBROUTINE COM_NEW_WINDOW(WNAME, COMM)
          CHARACTER(*), INTENT(IN) :: WNAME
          OPTIONAL, INTEGER, INTENT(IN) :: COMM
```

The `wName` argument is a character string and must be unique across all modules, and `comm` is the MPI communicator of the owner processes of the window. Because a window is a collective concept, this subroutine should be called on all processes within the MPI communicator. The communicator can be retrieved by calling `COM_get_communicator` (see subsection 3.5).

After a window is created, a user can create data attribute members and register addresses of data and functions to it as described later, followed by calling `COM_window_init_done` to mark the end of the registration of a window.

```
C++:    COM_window_init_done(const std::string name, int clct=1)
C:      COM_window_init_done(const char *name, int clct=1)
Fortran: SUBROUTINE COM_WINDOW_INIT_DONE(NAME, CLCT)
          CHARACTER(*), INTENT(IN) :: NAME
          INTEGER, INTENT(IN), OPTIONAL :: CLCT
```

It also takes the window name as its first argument. The second argument `clct` specifies whether the function is being called collectively on all the owner processes of the window, so that the mapping from panes to processes can be determined. If any pane was created or deleted, then `COM_window_init_done` must be called collectively before the window is used, by passing a non-zero value (the default) to `clct`. After calling `COM_window_init_done`, the sizes and arrays of the attributes can be changed without

calling `COM_window_init_done` again. Attributes and/or panes can be added or deleted, given that `COM_window_init_done` is called after the changes. If the attributes were changed but no panes were added or deleted from any process, then `COM_window_init_done` can be called with `clct` equal to zero, to avoid recomputing the pane-to-process mapping.

3.2.2 Data Attributes

Declaration of New Attributes Besides mesh data, a window can have other data members, which can be associated with the window, a pane, nodes, or elements of the pane. Different from keywords, these data attributes do not have designated names or data types. Therefore, a user must first define an attribute by calling `COM_new_attribute` before registering the addresses of the attribute. Again, this subroutine must be called collectively on all owner processes of a window.

```
C++:    COM_new_attribute(const std::string aName, char loc, COM_Type type,
           int ncomp, const char *unit)

C:      COM_new_attribute(const char *aName, char loc, COM_Type type,
           int ncomp, const char *unit)

Fortran: SUBROUTINE COM_NEW_ATTRIBUTE(ANAME, LOC, TYPE,
           NCOMP, UNIT)
           CHARACTER(*), INTENT(IN) :: ANAME, FNAME, UNIT
           CHARACTER*1, INTENT(IN) :: LOC
           INTEGER, INTENT(IN) :: TYPE, NCOMP
```

In the argument list, `aName` is the attribute name in the format of “window attribute”, similar to mesh data names; `loc` can be either `'w'`, `'p'`, `'n'`, or `'e'`, corresponding to *windowed*, *panel*, *nodal*, or *elemental* data; `type` specifies the base datatype of the attribute, which can be one of the constant defined in Appendix B; `ncomp` is the number of components of the attribute (for example, the number of entries associated with each node/element for nodal/elemental data); and `unit` is the unit of the data attribute, which can be the empty string “” if the attribute is unitless.

One can call `COM_new_attribute` on an existing attribute to re-define an attribute (including the keyword “nc”). However, one cannot increase the number of components of the predefined attributes (see next subsection). After calling `COM_new_attribute`, the previously registered data are reset, and its handles and inherited attributes become invalid. It is the user’s responsibility to ensure the consistency for the attribute.

Predefined Mesh Data Mesh data, including nodal coordinates, pane connectivity, and element connectivity (or simply connectivity), have predefined attribute names and data types. Nodal coordinates (“nc”) are predefined as double-precision nodal attributes with three components (corresponding to *x*, *y*, and *z*, respectively) per node and a default unit “m”. However, nodal coordinates may be redefined by calling `COM_new_attribute` to have less than 3 components, a different base data type, or a different unit. Pane connectivity (“pconn”) is predefined as a 1-D integer pane attribute with no unit.

“nc” for nodal coordinates
“pconn” for pane connectivity

For each pane, the pane-connectivity array can have multiple blocks:

1. shared nodes;
2. real nodes to send;
3. ghost nodes to receive;
4. real cells to send;
5. ghost cells to receive.

The first block goes into the real part of pconn and blocks 2–5 go into the ghost part. Blocks 2 and 3 must be present together, so are blocks 4 and 5. The ghost part of pconn is optional, and within the ghost part of pconn, blocks 4 and 5 are optional. Each block has the following content:

```

<number of communicating-pane blocks to follow>
<communicating pane id 1>
<#local nodes to follow>
<list of local nodes>...
... ! repeat for other remote panes

```

The lists of nodes for a pair of communicating panes are stored in the same order in their corresponding tables. Furthermore, the panes are stored in increasing order of pane IDs. If a node is shared by more than two panes, then every pair of shared nodes is stored in pconn. Note that it is possible for a single pane to have duplicated nodes, for example, in the case of branch-cut for structured meshes. In this case, in the block for shared nodes, the list of local nodes is composed of a series of node pairs, where the first node in the pair always has a smaller node ID than the second, and the number of local nodes is equal to twice the number of pairs. Note that in the case of partial inheritance, where a subwindow may inherit a subset of panes from a parent window, pconn may be inherited by the subwindow for its inter-pane communication, and as a consequence pconn may refer to some remote panes that no longer exist. In this case, it is important to note that the first number in each block is no longer the actual number of communicating panes, and a traversal of pconn should skip the nonexistent remote panes.

The names of element connectivities have the format of “:*elementtype:aname*”, where the “:*aname*” part is optional and is useful when there are multiple connectivity tables for one type of elements. Note that element connectivities are not regular attributes, in that different panes may contain different types of elements, and an element connectivity must not be created by calling COM_new_attribute but by setting its size and registering its address.

```

“:st1:aname”, “:st2:aname”, “:st3:aname” for structured mesh of 1, 2 and 3 dimensions.
“:b2:aname” and “:b3:aname” for 2- and 3-node bar elements.
“:t3:aname”, “:t6:aname”, “:q4:aname”, “:q8:aname”, and “:q9:aname” for connectivity tables of 3- and 6-node triangles, and 4-, 8-, and 9-node quadrilaterals, respectively.
“:T4:aname”, “:T10:aname”, “:B8:aname” (“:H8:aname”), and “:B20:aname” for connectivity tables of 4- and 10-node tetrahedra, and 8- and 20-node bricks, respectively
“:P5:aname”, “:P14:aname”, “:P6:aname” (“:W6:aname”), “:P15:aname” (“:W15:aname”), and “:P18:aname” (“:W18:aname”) for connectivity tables of 5- and 14-node pyramids and 6-, 15-, and 18-node prisms (aka pentahedra or wedges), respectively.

```

For elements of unstructured meshes, Roccom uses the same numbering convention as the CFD General Notation System (CGNS), of which a detailed description can be found in Section 3.3 of CGNS Standard Interface Data Structures (<http://www.grc.nasa.gov/WWW/cgns/sids/sids.pdf>). If a pane has multiple connectivity tables, these tables must be registered in increasing order of the element numbering (i.e., the elements with smaller indices in field-variable arrays must be registered earlier), and ghost elements must be registered last. Note that for structured meshes, a pane can register only one connectivity using `COM_set_array_const` described in the next subsection, by passing in the numbers of nodes of all directions in a single array listed in Fortran convention (See example code in subsection 3.2.4). We will allow for users to add new element types in future releases.

Registration of Sizes One sets the sizes of an attribute using the following routine.

```
C++:      COM_set_size(const std::string aName, int pane_id, int size, int ng=0)
C:        COM_set_size(const char *aName, int pane_id, int size, int ng=0)
Fortran:  SUBROUTINE COM_set_size(ANAME, PANE_ID, SIZE, NG)
          CHARACTER(*), INTENT(IN) :: ANAME
          INTEGER, INTENT(IN) :: PANE_ID, SIZE, NG
          OPTIONAL :: NG
```

In the arguments, `aName` is the data name in the format of “`window.aname`” or “`window.:elementtype:aname`” for connectivity tables `paneID` is a user-defined *positive* integer identifier of the pane, which must be unique within the window across all processors but need not be consecutive. Window attributes should be registered with pane-ID 0. The argument `size` is either the total number of nodes (including ghost nodes) in the pane (for nodal coordinates), or the number of elements (including ghost elements, for a connectivity table), or the length of the dataset for panel or windowed attributes. `ng` (optional in F90 and C++; default is 0) is either the number of ghost nodes in the pane for nodal data or the number of ghost elements for a connectivity table.

The default size of a window attribute is 1, but is undefined for other types of attributes. Note that setting the number of nodes for one nodal attribute affects all other nodal attributes, and it is more efficient to set size for “nc”. Typically, one should set the number of elements for each element connectivity.

Registration of Preallocated Array After creating an attribute, a user can register the address or addresses of the attribute using the following subroutine.

```
C++:      COM_set_array(const std::string aName, int paneID, void *addr,
          int stride=0, int cap=0)
C:        COM_set_array(const char *aName, int paneID, void *addr,
          int stride=0, int cap=0)
Fortran:  SUBROUTINE COM_SET_ARRAY(ANAME, PANEID, ADDR, STRIDE,
          CAP)
          CHARACTER(*), INTENT(IN) :: ANAME
          INTEGER, INTENT(IN) :: PANEID, STRIDE, CAP
          <TYPE> :: ADDR
          OPTIONAL :: STRIDE, CAP
```

As for `COM_set_size`, the `aName` is either an attribute name or the name to a connectivity table, and the `paneID` is a positive integer ID for the owner pane or 0 for window attributes. The `addr` argument specifies the address of the array for the attribute. If the components of each item are stored contiguously in an array of `Array(stride,cap)` in Fortran convention with `stride>=ncomp` and `cap>=size`, one can register the array by with a single call. If it is stored in an array of `Array(cap, ncomp)`, then the stride should be set to 1. The stride argument can be omitted if stride is equal to ncomp, and it is invalid if stride is greater than 1 but smaller than ncomp. **In Fortran 90, it is very important not to register a scalar variable defined locally in a subroutine or function (i.e., a stack variable), unless it has the TARGET or POINTER property.**

The `cap` argument can be omitted if `cap==size`. Otherwise, the user must register an array for each individual component of the attribute using attribute name in the format “window.i-attribute”, where *i* is an integer between 1 and the number of components of the attribute (Not applicable for connectivity tables). One can change the sizes and the arrays by calling `COM_set_size` and `COM_set_array`.

To protect data integrity, Roccom allows registration of a read-only data by calling `COM_set_array_const`, which takes the same arguments as `COM_set_array`.

```
C++:    COM_set_array_const(...)
C:      COM_set_array_const(...)
Fortran: SUBROUTINE COM_SET_ARRAY_CONST(...)
```

For Fortran 90, the types supported are scalars and pointers to 1-, 2-, and 3-dimensional integer, single-precision, and double-precision arrays. For other types of variables (such as a function pointer), one can register using one of the following two functions.

```
Fortran: SUBROUTINE COM_SET_EXTERNAL(ANAME, PANEID, VAR)
          CHARACTER(*), INTENT(IN) :: ANAME
          INTEGER, INTENT(IN) :: PANEID
          EXTERNAL VAR

Fortran: SUBROUTINE COM_SET_EXTERNAL_CONST(ANAME, PANEID, VAR)
```

One can obtain a pointer set by `COM_set_array_const` or `COM_SET_EXTERNAL` only through `COM_get_array_const`.

Registration of Bounds A user can register the lower and upper bounds of a specific attribute. One can register two sets of bounds: one set of hard bounds, which specifies the universal limits that the dataset must satisfy at all times and whose violation would result in runtime errors; the second set corresponds to soft bounds, whose violations would result in printing of warning messages at runtime.

```
C++:    COM_set_bounds(const std::string aName, int pane_id,
                    const void *lbnd, const void *ubnd, int is_soft=0)
C:      COM_set_bounds(const char *aName, int pane_id,
                    const void *lbnd, const void *ubnd, int is_soft=0)
Fortran: SUBROUTINE COM_set_bounds(ANAME, PANE_ID, LBND, UBND, IS_SOFT)
          CHARACTER(*), INTENT(IN) :: ANAME
          INTEGER, INTENT(IN) :: PANE_ID
          <TYPE>, INTENT(IN) :: LBND, UBND
          INTEGER, INTENT(IN), OPTIONAL :: IS_SOFT
```

When `pane_id` is 0, then the given bounds will be applied to all panes; if it is greater than 0, then they will be applied to the specific pane with the given pane ID. If the function is called for a vector attribute, then the bounds apply to the magnitude of the vectors. One can also set the bounds for individual components by calling the function on the corresponding component attributes (using attribute names “window.i-attribute”). The fifth argument, `is_soft`, which is optional with default value 0, specifies whether the bounds are hard (`is_soft==0`) or soft (`is_soft≠0`).

3.2.3 Functions

A window can contain not only data members but also function members. A function is registered into a window by calling `COM_set_function` on all owner processes of the window.

```
C:      COM_set_function(const char *fName, void (*faddr)(),
                        const char* intents, COM_Type types[])

Fortran: SUBROUTINE COM_SET_FUNCTION(FNAME, FADDR, INTENTS, TYPES)
          CHARACTER(*) , INTENT(IN) :: FNAME, INTENTS
          EXTERNAL FADDR
          INTEGER, INTENT(IN) :: TYPES
```

Similar to attribute names, `fName` has the format of “window.function”. The argument `faddr` takes the actual function pointer. For the C interface, to register a function that takes at least one argument (note that all arguments must be pointers), a user code must cast the pointer to the void `(*)()` type, which is predefined as `COM_Func_ptr`. The argument `intents` is a character string of length equal to the number of arguments taken by the registered function, and its *i*th character indicates whether the *i*th argument is for input, output, or both if `intentsi` is ‘i’/‘I’, ‘o’/‘O’, or ‘b’/‘B’, respectively (see the Optional Arguments paragraph of this section for more discussion). The argument `types` is an integer array of length also equal to the number of arguments, and its *i*th entry indicates the data type of the *i*th argument. All arguments of a registered function must be passed by reference. If a function is expecting an integer pointer/reference for its *i*th argument, for example, the *i*th entry should be either `COM_INT` (for C/C++) or `COM_INTEGER` (for Fortran). See Appendix B for a list of supported data types, and see the Data Type paragraph of this section for notes on special data types.

Member Function Many functions perform operations in a specific context. In object-oriented programs, such contexts are typically encapsulated in objects instead of being scattered into global variables as in traditional programs. Such an object is passed into a function as an argument, and frequently is passed implicitly by the compiler to allow cleaner interfaces in modern programming languages.

To encourage object-oriented programming and cleaner interfaces of application codes, Roccom supports the concept of member functions of attributes. A user registers a member function using the interface `COM_set_member_function`, which takes an attribute name as an additional argument.

```
C:      COM_set_member_function(const char *fName, void (*faddr)(),
                              const char* aName, const char* intents, COM_Type types[])

Fortran: SUBROUTINE COM_SET_MEMBER_FUNCTION(FNAME, FADDR,
      ANAME,
      INTENTS, TYPES)
```

```
CHARACTER(*), INTENT(IN) :: FNAME, ANAME, INTENTS
EXTERNAL FADDR
INTEGER, INTENT(IN) :: TYPES(:)
```

The given attribute should encapsulate the context of the registered function. The first entries in intents and types should specify the intention and data type of this attribute, respectively. When an application code invokes a registered member function through Roccom, it will not list this attribute in the arguments, but Roccom will pass it implicitly as the first argument to the function.

In addition, Roccom also provides a function for registering C++ member functions of a class, which must be a derived class of `COM_Object`. An object of a derived class of `COM_Object`, especially those with virtual functions, must be registered and retrieved using `COM_set_object()` and `COM_get_object()`, which takes the same arguments as `COM_set_array()` and `COM_get_array()`. The member functions are registered using the following interface,

```
C++:      COM_set_member_function(const char *fName, void (COM_Object::*faddr)(),
      const char* aName, const char* intents, COM_Type types[])
```

and must be casted to the `void (COM_Object::*faddr)()` type, which is predefined as `COM_Member_func_ptr`. For example, a member function `func` of a class `Rocfoo` can be casted as

```
reinterpret_cast<COM_Member_func_ptr>(&Rocfoo:func).
```

Optional Arguments If a registered function is written in C or C++, the last few arguments can be specified as *optional*. Roccom will pass in null pointers for them if the caller omit these arguments. To specify an argument to be optional, a user should use uppercase letters 'I', 'O', or 'B' instead of 'i', 'o', or 'b' in its corresponding entry in intents.

Data Types As we noted earlier, all arguments of a registered data must be passed by reference. A primitive data type (such as `COM_INT`) used in the argument types would indicate that the function is expecting a pointer or reference to that type. There are three special cases, however. First, if a function is expecting a character string (vs. a single character), which must be null terminated for C/C++ functions or whose length must be passed in implicitly for Fortran functions, then the corresponding data type of the argument must be set to `COM_STRING`. This data type tells Roccom to adapt the string if necessary (such as null-terminating the charactering string) to bridge C/C++ and Fortran transparently from users. Second, if a function is a service utility written in C++ and is expecting a C++ object that contains the description of an attribute, the corresponding data type of the argument must be set to `COM_METADATA`. If the function is expecting the physical address of a window attribute, then the corresponding datatype should be `COM_RAWDATA`. In general, two types of arguments should use `COM_RAWDATA`: the implicit argument for a member function, and an argument that is a function pointer.

Limitations and Special Notes For language interoperability, a registered function must return no value, and all its arguments must be passed by reference (i.e., must be pointers/references for C/C++ functions). Due to technical reasons, Roccom has to impose a limit on the maximum number of the arguments that a registered function can take, and the limit is currently set to 7, including the implicit arguments passed by Roccom, i.e., the first argument of member functions and character lengths for Fortran functions. This preset limit is large enough for most applications, but can be enlarged by changing Roccom's implementation if desired. Similar to data attributes, a function can be registered multiple times, but only the address of the last registration will be used.

3.2.4 Example Code

The following is a piece of Fortran code segment that demonstrates the registration of data and functions.

```
INTEGER          :: nn1, ni2, nj2    ! sizes of nodes
INTEGER          :: ne1              ! sizes of elements
INTEGER          :: types(2), dims(2)
INTEGER, POINTER :: conn1(3,ne1)
DOUBLE PRECISION, POINTER :: coors1(3,nn1), coor2(3,ni2, nj2)
DOUBLE PRECISION, POINTER :: disp1(3,nn1), disp2(3,ni2, nj2)
DOUBLE PRECISION, POINTER :: velo1(ne1,3), velo2(ni2-1, nj2-1,3)
EXTERNAL        fluid_update

CALL COM_NEW_WINDOW("fluid", MPI_COMM_WORLD)

! Create a node-centered double-precision dataset
CALL COM_NEW_ATTRIBUTE( "fluid.disp", "n", COM_DOUBLE, 3, "m")

! Create a element-centered double-precision dataset
CALL COM_NEW_ATTRIBUTE( "fluid.velo", "e", COM_DOUBLE, 3, "m/s")

! Create a pane with ID 11 of a triangular surface mesh
CALL COM_SET_SIZE("fluid.nc", 11, nn1)
CALL COM_SET_ARRAY("fluid.nc", 11, coors1, 3)
CALL COM_SET_SIZE("fluid.:t3:", 11, ne1)
CALL COM_SET_ARRAY("fluid.:t3:", 11, conn1, 3)

! Create a pane with ID 21 of a structured surface mesh
dims(1)=ni2; dims(2)=nj2;
CALL COM_SET_ARRAY_CONST("fluid.:st2:actual", 21, dims)
CALL COM_SET_ARRAY("fluid.nc", 21, coors2, 3)

! Register addresses of data attributes for both panes
CALL COM_SET_ARRAY("fluid.disp", 11, disp1)
CALL COM_SET_ARRAY("fluid.velo", 11, velo1, 1) ! Staggered layout
CALL COM_SET_ARRAY("fluid.disp", 21, disp2)
CALL COM_SET_ARRAY("fluid.velo", 21, velo2, 1) ! Staggered layout

! Register a function that takes two input arguments
type(1)=COM_DOUBLE; type(2)=COM_DOUBLE
CALL COM_SET_FUNCTION("fluid.update", fluid_update, "ii", types)

CALL COM_WINDOW_INIT_DONE("fluid")
.....
CALL COM_DELETE_WINDOW("fluid")
```

3.3 Procedure Calls

3.3.1 Attribute and Function Handles

A handle is an integer from which Roccom can obtain the actual data about attributes and functions. A user can obtain a mutable handle to an attribute using `COM_get_attribute_handle`, or an immutable handle using `COM_get_attribute_handle_const`, which take the same arguments.

```
C++:      int COM_get_attribute_handle(const std::string aName)
C:        int COM_get_attribute_handle(const char *aName)
Fortran:  FUNCTION COM_GET_ATTRIBUTE_HANDLE(ANAME)
          CHARACTER(*), INTENT(IN) :: ANAME
          INTEGER :: COM_get_attribute_handle
```

The function can be called on user-defined attributes, or a pre-defined attribute “nc”, “conn”, “pconn”, “mesh”, “pmesh”, “atts”, and “all”, which refer to nodal coordinates, element connectivity, pane connectivity, mesh data (including coordinates and element connectivity), parallel mesh data (including mesh and pane connectivity), all field attributes (excluding parallel mesh), and all data attributes, respectively. Note that it is illegal to call `COM_get_attribute_handle` on connectivity tables, whose scopes are within panes instead of within windows.

To obtain a handle to a function, one should use `COM_get_function_handle` instead, whose prototype is essentially the same as `COM_get_attribute_handle`.

If the function or attribute exists, then a positive integer ID will be returned; otherwise, 0 will be returned. So these functions can be used to detect the existence of a function or attributes. Similarly, one can detect the existence of a window by calling `COM_get_window_handle`.

3.3.2 Invocation

To invoke a function registered with Roccom in C or Fortran, a user need to use the following function.

```
C:        COM_call_function(int fHandle, int argc, void *arg1, ...)
Fortran:  SUBROUTINE COM_CALL_FUNCTION(FHANDLE, ARGC, ARG1, ...)
          INTEGER, INTENT(IN) :: FHANDLE, ARGC
          <TYPE> :: ARG1, ...
```

The first argument is a function handle, and the second attribute is the number of arguments to be passed, followed by the pointers (or references) to the data values or attribute handles.

For C++, we take advantage of the function overloading feature of the language to provide a cleaner interface `COM_call_function`.

```
C++:      COM_call_function(int fHandle, void *arg1, ...)
```

It does not require passing the number of arguments.

3.3.3 Call Tracing

To help debugging application codes, Roccom allows users to trace the procedure calls by setting a nonzero verbose level.

```
C:      COM_set_verbose(int v)
Fortran: SUBROUTINE COM_SET_VERBOSE(V)
         INTEGER, INTENT(IN) :: V
```

If v is a positive number, then Roccom will print out traces of the calls up to depth $(v + 1)/2$. If v is an odd number, Roccom will print only the names of the functions if v even, Roccom will also print the data types and values of the arguments passed to the functions.

3.3.4 High-Level Profiling

Roccom contains a simple profiling tool for timing the execution times of the functions invoked through Roccom.

```
C:      COM_set_profiling(int enable)
Fortran: SUBROUTINE COM_SET_PROFILING(ENABLE)
         INTEGER, INTENT(IN) :: ENABLE
```

If `enable` is zero, it disables profiling; otherwise, it enables profiling and resets all the counters of the profiler.

In a parallel run, the timing results are typically more accurate if `MPI_Barrier` is called before and after a function call, but putting too many barriers may also affect performance. Roccom allows a user to control where barriers should be placed by the following call.

```
C:      COM_set_profiling_barrier(int fHandle, MPI_Comm comm)
Fortran: SUBROUTINE COM_SET_PROFILING_BARRIER(FHANDLE, COMM)
         INTEGER, INTENT(IN) :: FHANDLE, COMM
```

This routine will enable Roccom to call `MPI_Barrier` on the given communicator before and after the given function for the processes of the given communicator.

The profiling results can be printed by calling

```
C:      COM_print_profile(const char *fname, const char *header)
Fortran: SUBROUTINE COM_PRINT_PROFILE(FNAME, HEADER)
         CHARACTER(*), INTENT(IN) :: FNAME, HEADER
```

This routine will append the header and the timing results to the file with name `fname`. If `fname` is `NULL` or the empty string, the standard output will be used instead. A typical timing result looks as follows.

Function	#calls	Time(tree)	Time(self)
Rocflu.update_solution	100	43.1856	42.8221
Rocfrac.update_solution	100	30.4038	30.3861
RFC.least_squares_transfer	400	0.957599	0.957599
.....			
Total(top level calls)			74.806

In the output, the Time(tree) indicates the sum of the elapsed wall-clock time during the execution of a function since the last call to `COM_init_profiling`, and Time(self) subtracts the elapsed time of the calls made with the function.

3.3.5 Calling System Calls in Fortran

To allow Fortran to execute a shell command using the system call interface of C, Roccom provides the following `COM_CALL_SYSTEM` function.

```
Fortran:  FUNCTION COM_CALL_SYSTEM(COMMAND)
          CHARACTER(*), INTENT(IN) :: COMMAND
```

It will execute the command and return the return status of the command after the command has been completed; if the command fails to execute due to fork failure, then -1 will be returned.

3.3.6 Calling AtExit and Exit Functions In Fortran

A Fortran code can also call the `atexit` and `exit` functions of the C standard through Roccom.

```
Fortran:  FUNCTION COM_CALL_ATEXIT(FUNC)
          EXTERNAL FUNC
```

```
Fortran:  FUNCTION COM_CALL_EXIT(IERR)
          INTEGER, INTENT(IN) :: IERR
```

`COM_CALL_ATEXIT` registers a subroutine to be executed when the program terminates normally. `COM_CALL_EXIT` causes the program to end and supplies a status code to the calling environment.

3.4 Advanced Window Management

3.4.1 Memory Management

Sometimes, it is more convenient to let Roccom allocate arrays instead of registering user-allocated arrays. This approach avoids having to duplicate the data structures of windows and panes in application codes for multi-block meshes, and is particularly beneficial for implementing complex orchestration modules (such as Rocman in GEN2.5). Roccom provides the following subroutines for memory allocation.

```

C++:    COM_allocate_array(const std::string aName, int panelD=0, void **addr=NULL,
int strd=0, int cap=0)

C:      COM_allocate_array(const char *aName, int panelD=0, void **addr=NULL,
int strd=0, int cap=0)

Fortran: SUBROUTINE COM_ALLOCATE_ARRAY(ANAME, PANEID, ADDR,
STRIDE, CAP)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANEID, STRIDE, CAP
<TYPE>, POINTER :: ADDR
OPTIONAL :: PANEID, ADDR, STRIDE, CAP

C++:    COM_resize_array(const std::string aName, int panelD=0, void **addr=NULL,
int stride=-1, int cap=0)

C:      COM_resize_array(const char *aName, int panelD=0, void **addr=NULL,
int stride=-1, int cap=0)

Fortran: SUBROUTINE COM_RESIZE_ARRAY(ANAME, PANEID, ADDR,
STRIDE, CAP)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANEID, STRIDE, CAP
<TYPE>, POINTER :: ADDR
OPTIONAL :: PANEID, ADDR, CAP, STRIDE

```

These functions take arguments similar to **COM_set_array**, except that `addr` is returned passed out instead of passed into the procedure. They allocate memory for a specific attribute in a given pane if `panelD` is nonzero or all panes if `panelD` is zero (the default value). The differences between **allocate** and **resize** are that the latter allocates memory only if the array was not yet initialized, or was previously allocated by Roccom but the current capacity is increased or the stride is no longer the same. During **resize**, values of the old array will be copied automatically to the new array. If `strd` is -1, which is the default for **COM_resize_array**, the current value registered with Roccom (or the number of components if not yet registered) will be used; if `strd` is 0, then the number of components of the attribute will be used. If `cap` is 0, then the larger of the current capacity and the number of items will be used. Note that it is an error to **resize** an inherited or user-allocated (i.e., not allocated by Roccom) attribute. For the Fortran interface, only scalar, 1-D and 2-D pointers are allowed. If a scalar pointer is used, the data itself must be a scalar and the argument `STRD` and `CAP` must not be present. If a 1-D pointer is given, then the size of the array will be `STRD*CAP`. If a 2-D pointer is given, then the sizes of the array will be `(STRD,CAP)` if `STRD` is no smaller than the number of components of the attribute (`NCOMP`), or be `(CAP,NCOMP)` if `STRD` is 1.

A user can use the keyword **all** in the form of “`window.all`” for `aName` to have Roccom allocate memory for all attributes (including the mesh) in a window. The capacity must be no smaller than the size specified by **COM_set_size**; if a value smaller than the actual size is passed to **COM_resize_array**, then the actual size will be used instead.

Furthermore, using **COM_append_array**, Roccom provides a function to append a series of values to the end of an array associated with a pane or window attribute that has no ghost items.

```

C++:    COM_append_array(const std::string aName, int panelD, const void *addr,
int strd, int size)

C:      COM_append_array(const char *aName, int panelD, const void *addr,

```

int strd, int size)

Fortran: **SUBROUTINE COM_APPEND_ARRAY**(ANAME, PANEID, ADDR,
STRD, SIZE)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANEID, STRD, SIZE
<TYPE>, INTENT(IN) :: ADDR

This function is equivalent to calling **COM_resize_array** to increase the capacity of the array if necessary using the stride currently registered with Rocom, calling **COM_set_size** to increase the number of items by size, and then copying data from user buffer addr with a stride strd. This function is particularly useful for packing a series of values into a big array in Rocom. Note that after calling **COM_append_array**, the array in Rocom may have been reallocated if its capacity was increased, in which case the address previously obtained from Rocom becomes invalid and the user must reobtain the address by calling **COM_get_array**.

Allocated memory should be deallocated by calling **COM_deallocate_array**.

C++: **COM_deallocate_array**(const std::string aName, int panelD=0)
C: **COM_deallocate_array**(const char *aName, int panelD=0)
Fortran: **SUBROUTINE COM_DEALLOCATE_ARRAY**(ANAME, PANEID)
CHARACTER(*), INTENT(IN) :: ANAME
INTEGER, INTENT(IN) :: PANEID
OPTIONAL :: PANEID

If the deallocation routine is not called, the memory will be freed automatically when the window is destroyed.

3.4.2 Pointer Attributes

Rocom provides two special data types, **COM_VOID** and **COM_F90POINTER**. The former means a void pointer in C or C++, and the latter a Fortran 90 pointer. A F90 pointer is different from C/C++ pointers, in that it is a structure containing the descriptor of the data that are referenced, and the exact size of the structure is compiler dependent and may vary with the types that it references. These two data types are particularly useful in conjunction with **COM_allocate_array** to store pointers to some objects, which allows a module to eliminate global variables completely, so that they can take advantage of Charm++.

When Rocom allocates a F90 pointer, it allocates a piece of memory that is large enough to hold any type of F90 pointers. A F90 application code can copy a pointer to or from Rocom using **COM_SET_POINTER** and **COM_GET_POINTER**, respectively.

Fortran: **SUBROUTINE COM_SET_POINTER**(ATTR, PTR, ASSO)
CHARACTER(*), INTENT(IN) :: ATTR
<TYPE>, POINTER :: PTR
EXTERNAL ASSO

Fortran: **SUBROUTINE COM_GET_POINTER**(ATTR, PTR, ASSO)
CHARACTER(*), INTENT(IN) :: ATTR
<TYPE>, POINTER :: PTR
EXTERNAL ASSO

These functions are particularly useful for registering the context variable of member functions, similar to registering COM_Object associated with the C++ member functions. For that reason, Roccom also provides two F90 interface functions, `COM_set_object` and `COM_get_object`, which are essentially aliases of `COM_set_pointer` and `COM_get_pointer`. The argument ASSO is a user-defined subroutine which looks like follows.

```

SUBROUTINE ASSOCIATE_POINTER( attr, ptr)
  <TYPE>, POINTER  :: attr
  <TYPE>, POINTER  :: ptr

  ptr => attr
END SUBROUTINE ASSOCIATE_POINTER

```

Because the arguments of `COM_set_pointer` and `COM_get_pointer` are pointers whose types are unknown to Roccom, the user must explicitly define the prototypes of these functions in the application codes using the specific data types.

3.4.3 Inheritance

Inheritance is a key concept of object-oriented programming. In GEN2.5, inheritance is very useful under a few situations. First, the orchestration module (Rocman) sometimes needs to create intermediate data associated with a window owned by another module. Inheritance allows Rocman to extend the window by adding additional data attributes, or altering the definitions of some of the attributes. Second, a module (e.g., Rocburn) may need to operate on a subset of the mesh of another module (e.g., Rocflo or Rocflu). Roccom facilitates such special needs by allowing a window to inherit (a subset of) another window without incurring the memory overhead of data duplication. Furthermore, Rocman sometimes needs to split user-defined windows into separate windows based on boundary-condition types, so that they can be handled differently (such as written into separate files for visualization).

Roccom supports two types of inheritance: using and cloning. For the former, Roccom does not duplicate the dataset; for the latter, Roccom does. For each type, it allows inheriting the mesh from a parent window to a child window in two modes. First, the mesh can be inherited as a whole. Second, only a subset of panes that satisfy a certain criterion are inherited. The following two subroutines support these two modes of use-inheritance, respectively.

```

C++:      COM_use_attribute(const std::string wName_to, const std::string wName_from,
                          int with_ghost=1, const char *aName=NULL, int val=0)

C:        COM_use_attribute(const char *wName_to, const char *wName_from,
                          int with_ghost=1, const char *aName=NULL, int val=0)

Fortran:  SUBROUTINE COM_USE_ATTRIBUTE(WNAME_TO, WNAME_FROM,
                          WITH_GHOST, ANAME, VAL)
          CHARACTER(*), INTENT(IN) :: WNAME_TO, WNAME_FROM, ANAME

```

INTEGER, INTENT(IN) :: VAL, WITH_GHOST
OPTIONAL WITH_GHOST, ANAME, VAL

C++: **COM_clone_attribute**(const std::string wName_to, const std::string wName_from,
int with_ghost=1, const char *aName=NULL, int val=0)

C: **COM_clone_attribute**(const char *wName_to, const char *wName_from,
int with_ghost=1, const char *aName=NULL, int val=0)

Fortran: **SUBROUTINE COM_CLONE_ATTRIBUTE**(WNAME_TO, WNAME_FROM,
WITH_GHOST, ANAME, VAL)
CHARACTER(*), INTENT(IN) :: WNAME_TO, WNAME_FROM, ANAME
INTEGER, INTENT(IN) :: VAL, WITH_GHOST
OPTIONAL WITH_GHOST, ANAME, VAL

In the arguments, the wName are window names and the aName are attribute names. The argument with_ghost indicates whether the ghost nodes and elements should be inherited. The next argument is a panel attribute of integer type, and only the panes whose corresponding values of the attribute equal to the argument val will be inherited. In practice, aName is most likely to correspond to a boundary-condition type for panes, and val correspond to a boundary condition ID. Note that if aName is empty (i.e., either a NULL pointer or an empty string) and val is nonzero, then condition is considered to be “paneID==val”, so that only the pane whose ID is equal to val is inherited.

If a child window needs to contain panes of more than one boundary-condition types, then a user can call **COM_use_attribute** multiple times with different boundary condition ID. Note that in both routines, the child window does not duplicate memory space for the mesh but inherit the memory addresses of the parent window. If a pane in the parent window does not exist in the target window, a new pane is inserted into the derived window if the attribute being inherited contains the element connectivities (i.e., “conn”, “mesh”, or “all”), or the pane is ignored for other types of elements. If the attribute being inherited already exists in the child window, then the data type and layout of the new attribute must be the same as the existing one, and Rocom will overwrite other information of the existing attribute with the new attribute. Note that one must not delete or redefine an attribute that is being used by another window.

A related function of inheritance is **COM_copy_attribute**, which copies data from one attribute onto another.

C++: **COM_copy_attribute**(const std::string wName_to, const std::string wName_from,
int with_ghost=1, const char *aName=NULL, int val=0)

C++: **COM_copy_attribute**(const int wName_to, const int wName_from,
int with_ghost=1, const char *aName=NULL, int val=0)

C: **COM_copy_attribute**(const char *wName_to, const char *wName_from,
int with_ghost=1, const char *aName=NULL, int val=0)

Fortran: **SUBROUTINE COM_COPY_ATTRIBUTE**(WNAME_TO, WNAME_FROM,
WITH_GHOST, ANAME, VAL)
CHARACTER(*), INTENT(IN) :: WNAME_TO, WNAME_FROM, ANAME
INTEGER, INTENT(IN) :: VAL, WITH_GHOST
OPTIONAL WITH_GHOST, ANAME, VAL

3.4.4 Deletion of Entities

When a window is not needed anymore, it should be destroyed by calling `COM_delete_window`, which takes the window name as its only argument.

```
C:      COM_delete_window(const std::string wName)
C:      COM_delete_window(const char *wName)
Fortran: SUBROUTINE COM_DELETE_WINDOW(WNAME)
         CHARACTER(*), INTENT(IN) :: WNAME
```

This subroutine allows Roccom to clean up its internal data created for a window. It also automatically deallocates all the datasets allocated using `COM_allocate_array` or `COM_resize_array` but not yet deallocated.

Furthermore, one can delete a single pane from a window by calling `COM_delete_pane`, which takes the window name and a pane ID as its arguments.

```
C++:    COM_delete_pane(const std::string wName, int pandID)
C:      COM_delete_pane(const char *wName, int pandID)
Fortran: SUBROUTINE COM_DELETE_PANE(WNAME, PANEID)
         CHARACTER(*), INTENT(IN) :: WNAME
         INTEGER, INTENT(IN) :: PANEID
```

One can also delete an existing attribute (except for predefined attributes) by calling `COM_delete_attribute`.

```
C++:    COM_delete_attribute(const std::string aName)
C:      COM_delete_attribute(const char *aName)
Fortran: SUBROUTINE COM_DELETE_ATTRIBUTE(ANAME)
         CHARACTER(*), INTENT(IN) :: ANAME
```

The only keyword that can be used with `COM_delete_attribute` is “atts”, which will removed all user-defined attributes and leave only the mesh. Note that after deleting some panes or attributes, one must call `COM_window_init_done` on all processes collectively before using the window. In addition, deleting a window, pane, or attribute may invalidate attribute and function handles and the structure of inheritance, so they should be used with extreme care.

3.5 Information Retrieval

3.5.1 Window and panes

Typically, data registered by application modules need to be accessed only by service modules through the C++ interface described in the Developers Guide. However, some application modules (e.g., Rocburn) need to obtain the information about a window created by another module (e.g., Rocflo/Rocflu). Roccom 3 provides functions to support information retrieval, under the assumption that the caller knows about the attribute names and base data types of the attributes.

```

C:      COM_get_communicator(const char *wName, MPI_Comm *comm)
Fortran: SUBROUTINE COM_GET_COMMUNICATOR(WNAME, COMM)
         CHARACTER(*), INTENT(IN) :: WNAME
         INTEGER, INTENT(OUT) :: COMM

```

This subroutine obtains the MPI communicator of a window.

The following subroutine obtains the IDs of the panes in a window local to a process:

```

C/C++:   COM_get_panes(const char *wName, int *np, int **pane_ids=NULL,
                    int rank=myrank)
C++:     COM_get_panes(const char *wName, vector<int> &pane_ids, int rank=myrank)
Fortran:  SUBROUTINE COM_GET_PANES(WNAME, NP, PANE_IDS, RANK)
         CHARACTER(*), INTENT(IN) :: WNAME
         INTEGER, INTENT(OUT) :: NP
         OPTIONAL, INTEGER, POINTER :: PANE_IDS(:)
         OPTIONAL, INTEGER, INTENT(IN) :: RANK

```

It sets the number of panes to np and loads an array of IDs into pane_ids, whose memory is allocated by Rocomm and should be deallocated by calling **COM_free_buffer** (except for the vector interface). The rank is in the scope of the MPI communicator of the window. If the rank is not present or is -2, then the default value is that of the current process. If the rank is -1, then the function will load the panes on all the processes within the communicator. Note that this function can only be called **after** calling **COM_window_init_done**.

```

C/C++:   COM_get_attributes(const char *wName, int *na, char **names)
C++:     COM_get_attributes(const char *wName, int *na, string &names)
Fortran:  SUBROUTINE COM_GET_ATTRIBUTES(WNAME, NA, NAMES)
         CHARACTER(*), INTENT(IN) :: WNAME
         INTEGER, INTENT(OUT) :: NA
         CHARACTER, POINTER :: NAMES(:)

```

It sets na to be the number of attributes in the window and allocates a space-delimited string names to store the names of the attributes. Except for the string interface, names must be deallocated by calling **COM_free_buffer** (see below) after use.

```

C/C++:   COM_get_connectivities(const char *wName, const int *pid,
                    int *nc, char **names)
C++:     COM_get_connectivities(const char *wName, const int *pid,
                    int *nc, string &names)
Fortran:  SUBROUTINE COM_GET_CONNECTIVITIES(WNAME, PID, NC, NAMES)
         CHARACTER(*), INTENT(IN) :: WNAME
         INTEGER, INTENT(IN) :: PID
         INTEGER, INTENT(OUT) :: NC
         CHARACTER, POINTER :: NAMES(:)

```

It sets `nc` to be the number of connectivity tables in a pane and allocates a space-delimited string names to store the names of the connectivity tables. Again, names must be deallocated by calling `COM_free_buffer` (except for the string interface) after use.

```
C:      COM_free_buffer( char (or int) **buf)
Fortran: SUBROUTINE COM_FREE_BUFFER( BUF)
         CHARACTER (or INTEGER), POINTER :: BUF(:)
```

3.5.2 Attribute and Connectivity

One can obtain the information of an attribute by calling `COM_get_attribute`, whose arguments correspond to those of `COM_new_attribute`.

```
C:      COM_get_attribute(const char *aName, char *loc, COM_Type *type,
                        int *ncomp, char *unit, int n)
C++:   COM_get_attribute(const char *aName, char *loc, COM_Type *type,
                        int *ncomp, string *unit)
Fortran: SUBROUTINE COM_GET_ATTRIBUTE(ANAME, LOC, TYPE,
                                     NCOMP, UNIT)
         CHARACTER(*), INTENT(IN) :: ANAME
         CHARACTER*1, INTENT(OUT) :: LOC
         INTEGER, INTENT(OUT) :: TYPE, NCOMP
         CHARACTER(*) :: UNIT
```

One can also use `COM_get_attribute` on a connectivity, which will set `ncomp` to the number of nodes per element for that particular type of element.

One can also check the status of a window, a pane, an attribute, or a connectivity table by calling the function `COM_get_status`.

```
C:      int COM_get_status(const char *aName, int paneID)
Fortran: FUNCTION COM_GET_STATUS(ANAME, PANEID)
         CHARACTER(*), INTENT(IN) :: ANAME
         INTEGER, INTENT(IN) :: PANEID
         INTEGER :: COM_GET_STATUS
```

If `aName` is a window name (i.e., containing no `'`) and `paneID` is 0, then it checks whether the window exists, and returns 0 if so and -1 otherwise. If `aName` is a window name and `paneID` is nonzero, then it checks whether the given pane exist in the window, and returns 0 if so and -1 otherwise. If `aName` is in the form of “window.attribute”, then it checks the status of the given attribute, and returns one of the following values:

- -1: does not exist;
- 0: exists but not initialized;

- 1: set by the user using `set_array` or `set_object`;
- 2: set by the user using `set_array_const`;
- 3: use from another attribute;
- 4: allocated using `resize_array`, `allocate_array`, or cloned from another attribute.

If an attribute uses another, one can get the full name (`window.attribute`) of its parent attribute using the following interface:

```
C/C++:      COM_get_parent(const char *waName, int paneid, char **parent)
C++:        COM_get_parent(const char *waName, int paneid, string &parent)
Fortran:    SUBROUTINE COM_GET_PARENT(WANAME, PANEID, PARENT)
            CHARACTER(*) , INTENT(IN) :: WANAME
            INTEGER, INTENT(IN) :: PANEID
            CHARACTER, POINTER :: PARENT(:)
```

The storage for the parent name will be allocated by `Rocom` and must be freed using `COM_free_buffer` after usage, except for the C++ interface.

3.5.3 Sizes

The following function can be used to obtain the sizes of an attribute. Note that the size corresponds to the total size (including ghost items).

```
C++:      COM_get_size(const std::string aName, int pane_id, int *size, int *ng=NULL)
C:        COM_get_size(const char *aName, int pane_id, int *size, int *ng=NULL)
Fortran:  SUBROUTINE COM_get_size(ANAME, PANE_ID, SIZE, NG)
            CHARACTER(*) , INTENT(IN) :: ANAME
            INTEGER, INTENT(IN) :: PANE_ID
            INTEGER, INTENT(OUT) :: SIZE, NG
            OPTIONAL :: NG
```

Note that for structured meshes, its dimensions should be obtained using `COM_get_array_const` instead of `COM_get_size`.

3.5.4 Arrays

One can obtain an array by either obtaining a pointer to the array, or copying the data into a user provided buffer. The first mode is provided by `COM_get_array` and `COM_get_array_const`, which can be used to obtain a pointer to an array registered or allocated in *Rocom*.

```

C:      COM_get_array(const char *aName, int panelID, void **addr,
                    int *strd=NULL, int *cap=NULL)

Fortran: SUBROUTINE COM_GET_ARRAY(ANAME, PANEID, ADDR,
                                STRD, CAP)
        CHARACTER(*), INTENT(IN) :: ANAME
        INTEGER, INTENT(IN) :: PANEID
        <TYPE>, POINTER :: ADDR
        INTEGER, INTENT(OUT) :: STRD, CAP
        OPTIONAL :: STRD, CAP

C:      COM_get_array_const(...)

Fortran: SUBROUTINE COM_GET_ARRAY_CONST(...)

```

Note that if the attribute-name and the pane-ID do not identify a unique array, then a NULL pointer will be returned. Furthermore, if an array was registered with **COM_set_array_const**, then it can be retrieved only by **COM_get_array_const**. For the Fortran interface, only scalar, 1-D and 2-D pointers are allowed. If a scalar pointer is used, the data itself must be a scalar and the argument STRD and CAP must not be present. If a 1-D pointer is given, then the size of the array will be STRD*CAP. If a 2-D pointer is given, then the sizes of the array will be (STRD,CAP) if STRD is no smaller than the number of components of the attribute (NCOMP), or be (CAP,NCOMP) if STRD is 1.

The second mode is provided by **COM_copy_array**.

```

C:      COM_copy_array(const char *aName, int panelID, void *val, int v_strd=0,
                    int v_size=0, int offset=0)

Fortran: SUBROUTINE COM_COPY_ARRAY(ANAME, PANEID, VAL, V_STRD,
                                V_SIZE, OFFSET)
        CHARACTER(*), INTENT(IN) :: ANAME
        INTEGER, INTENT(IN) :: PANEID, V_STRD, V_SIZE, OFFSET
        <TYPE>, POINTER :: VAL
        OPTIONAL :: V_STRD, V_SIZE, OFFSET

```

It copies up to v_size items of the attribute starting from the offset-th item of the attribute into the given buffer with stride v_strd. If v_strd=0 (the default value), then the number of components will be used as the stride. If v_size=0 (the default value), then the number of items will be used as the size. The default value of offset is 0. Note that a runtime error occurs if offset is negative or offset+v_size is larger than the actual capacity of the attribute.

3.5.5 Bounds

The lower and upper bounds of a specific attribute can be obtained by calling **COM_get_bounds**.

```

C:      COM_get_bounds(const char *aName, int pane_id,
                    void *lbnd, void *ubnd, int is_soft=0)

Fortran: SUBROUTINE COM_GET_BOUNDS(ANAME, PANE_ID, LBND, UBND, IS_HARD)
        CHARACTER(*), INTENT(IN) :: ANAME

```

```

INTEGER, INTENT(IN) :: PANE_ID
<TYPE>, INTENT(OUT) :: LBND, UBND
INTEGER, INTENT(IN), OPTIONAL :: IS_SOFT

```

Furthermore, Roccom also provides functions to check the attributes against pre-set bounds.

```

C:      int COM_check_bounds(const char *aName, int pane_id, int nprint=0)
Fortran: INTEGER FUNCTION COM_check_bounds(ANAME, PANE_ID, NPRINT)
          CHARACTER(*), INTENT(IN) :: ANAME
          INTEGER, INTENT(IN) :: PANE_ID
          INTEGER, INTENT(IN), OPTIONAL :: NPRINT

```

If `pane_id` is 0, then the bounds will be checked on all panes; otherwise, they will be checked only on the pane with the given pane ID. If there are only soft-bound violations or no violations, then the function returns the number of soft-bound violations. If the verbose level of Roccom is 0, then no information will be printed. If the verbose level is nonzero and `nprint` is 0, then a summary of soft-bound violations will be printed. If `nprint` is greater than 0, then the first few (where the number to be printed is `nprint`) soft-bound violations for each attribute in each individual pane will be printed. A violation of hard bounds will terminate the execution of the code, and a summary of hard-bound violations will be printed upon termination, along with information about any soft-bound violations if the verbose level is nonzero.

4 Makefiles for Roccom Applications

Writing makefiles is not a trivial task for complex codes. Besides defining all the dependencies between files, the makefile must specify the correct directory for Roccom header files, directories of the additional header files or libraries required by service utilities, and platform-dependent compiler options. Mixing C/C++ and Fortran codes together further complicates the makefiles.

These issues are apparently faced by all application codes. To maximize the reuse of makefiles, we provide a file `Makefile.common` in Roccom's root directory. (Thanks to Orion S. Lawlor for initiating the reorganization of makefiles.) The makefile of an application should include this common makefile and typically looks as follows.

Gengbin Zheng further extended the makefiles to use `autoconfig` which allows Rocstar to be built for multiple versions with one code base.

```

# Makefile file RocX
COMHOME      = ../Roccom include
$(COMHOME)/Makefile.common

VPATH        = src
LIBX         = libX.a
CPPFLAGS     += -Iinclude

```

```

OBJS                = foo.o

#===== Actions =====
all:                $(LIBX)

$(LIBX):            $(OBJS)
                   $(AR) $(ARFLAGS) $@ $?

```

These makefiles must be invoked using gmake because the common makefile uses some advanced features that are gmake specific.

5 Sample Codes

A few sample application codes of Roccom are provided in the test subdirectories of a few service modules. In particular, both C++ and Fortran 90 sample codes are available in the Roccom/Rochdf/test subdirectory. These codes demonstrate most functionality described in this document, including how to create windows, register data attributes and functions, duplicate windows using inheritance, allocate/deallocate memory in Roccom, and make procedure calls. The makefile in Roccom/Rochdf is also a good example for typical application or service modules. Similar sample codes are also available in Roccom/Rocblas/test and Roccom/Rocmap/test.

A Guide for Migrating to Roccom 3.x

This section provides a guide for mapping Roccom 2.x's interface to Roccom 3.x.

A.1 Mesh data

A.1.1 Definition of mesh unit

Old: `COM_init_mesh("win.unit", 0, "meter", 5)`
 New: `COM_new_attribute("win.nc", 'n', COM_DOUBLE, 3, "meter")`

A.1.2 Setting coordinates

Old: `COM_init_mesh("win.nc", pane_id, coors, num_nodes [, num_ghost_nodes])`
 New: `COM_set_size("win.nc", pane_id, num_nodes [, num_ghost_nodes])`
 `COM_set_array("win.nc", pane_id, coors [, stride, capacity])`
 Old: `COM_init_mesh("win.n-c", pane_id, coors, num_nodes [, num_ghost_nodes])` // "n-c"
 indicates staggered data layout.

New: `COM_set_size("win.nc", pane_id, num_nodes [, num_ghost_nodes])`
`COM_set_array("win.nc", pane_id, coors, 1 [, capacity])` // stride of 1 indicates the data layout.

A.1.3 Element connectivity

Old: `COM_init_mesh("win.t3", pane_id, elems, num_elems [, num_ghost_elems])`
New: `COM_set_size("win.:t3:aname", pane_id, num_elems [, num_ghost_elems])`
`COM_set_array("win.:t3:aname", pane_id, elems [, stride, capacity])`
Old: `COM_init_mesh("win.t-3", pane_id, elems, num_elems [, num_ghost_elems])`
New: `COM_set_size("win.:t3:aname", pane_id, num_elems [, num_ghost_elems])`
`COM_set_array("win.:t3:aname", pane_id, elems, 1 [, capacity])`

A.1.4 Structured Mesh

Old: `COM_init_mesh("win.st", pane_id, sizes, 2, ghost_layers)`
New: `COM_set_size("win.aname", 2, ghost_layers)`
`COM_set_array_const("win.:st2:aname", pane_id, sizes)`

For three dimensional meshes, the attribute name is `":st3:aname"`.

A.1.5 Inheritance of Mesh

Old: `COM_use_mesh("win", "parent")`
New: `COM_use_attribute("win.mesh", "parent.mesh")`
Old: `COM_clone_mesh("win", "parent")`
New: `COM_clone_attribute("win.mesh", "parent.mesh")`
Old: `COM_use_mesh_sub("win", "parent", with_ghost, "parent.cond", val)`
New: `COM_use_attribute("win.mesh", "parent.mesh", with_ghost, "parent.cond", val)`
Old: `COM_clone_mesh_sub("win", "parent", with_ghost, "parent.cond", val)`
New: `COM_clone_attribute("win.mesh", "parent.mesh", with_ghost, "parent.cond", val)`

A.2 Data Attributes

A.2.1 Nodal and Elemental Attributes

Old: `COM_init_attribute("win.aname", pane_id, addr)`
New: `COM_set_array("win.aname", pane_id, addr)`
Old: `COM_init_attribute_strided("win.aname", pane_id, addr, strd)`
New: `COM_set_array("win.aname", pane_id, addr, strd)`

A.2.2 External Fortran Symbols

Old: COM_INIT_ATTRIBUTE(“win.func”, 0, subroutine_name)

New: COM_SET_EXTERNAL(“win.func”, 0, subroutine_name)

A.2.3 Memory Allocation

Old: COM_allocate_attribute(“win.aname”, pane_id)

New: COM_allocate_array(“win.aname”, pane_id), or
COM_resize_array(“win.aname”, pane_id)

A.2.4 Information Retrieval

Old: COM_get_attribute_addr(“win.aname”, pane_id, addr)

New: COM_get_array(“win.aname”, pane_id, addr)

Old: COM_get_window_npanes(“win”, npanes)

New: COM_get_panes(“win”, npanes)

Old: COM_get_window_nnodes(“win”, npanes, nnodes_allpanes)

New: COM_get_size(“win.nc”, pane_id, nnodes) [Loop through all panes]

Old: COM_get_window_nelems(“win”, npanes, nelems_allpanes)

New: COM_get_size(“win.conn”, pid, nelems) [Loop through all panes]

A.3 Name Changes

The following names were changed:

COM_create_window → COM_new_window

COM_init_function → COM_set_function

COM_init_member_function → COM_set_member_function.

COM_init_profiling → COM_set_profiling

COM_init_profiling_barrier → COM_set_profiling_barrier

COM_get_attribute_info → COM_get_attribute

COM_get_window_panes → COM_get_panes

A.4 Changes Developers Interfaces

Changed Attribute::size() to Attribute::size_of_components().

Structured meshes also have a Connectivity object.

B Rocom Data types

Rocom defines the following primitive C data types:

COM_CHAR, COM_UNSIGNED_CHAR, COM_BYTE, COM_SHORT, COM_UNSIGNED_SHORT,
COM_INT, COM_UNSIGNED, COM_LONG, COM_UNSIGNED_LONG, COM_FLOAT, COM_DOUBLE,
COM_LONG_DOUBLE, COM_LONG_LONG_INT.

It also defines the following primitive Fortran data types

COM_CHARACTER, COM_LOGICAL, COM_INTEGER, COM_REAL, COM_DOUBLE_PRECISION,
COM_COMPLEX, COM_DOUBLE_COMPLEX.

The following C and Fortran data types are equivalent. In a heterogeneous application that requires interoperability of C/C++ and Fortran, it is advantageous to use only these primitive data types.

COM_CHAR \approx COM_CHARACTER
COM_INT \approx COM_INTEGER
COM_FLOAT \approx COM_REAL
COM_DOUBLE \approx COM_DOUBLE_PRECISION

There are also four special data types:

COM_STRING, COM_RAWDATA, COM_METADATA, COM_VOID, COM_F90POINTER.

Index

- adaptivity, 2
- API, 2, 8, 10
- argument
 - implicit, 19
 - intention, 18, 19
 - 'B', 19
 - 'I', 19
 - 'O', 19
 - 'b', 19
 - 'i', 19
 - 'o', 19
 - optional, 13, 16, 24, 25, 27, 29, 31, 32
- array, 16
 - allocate, 3, 6, 23, 24, 28, 29, 31
 - capacity, 3, 24
 - convention, 17
 - deallocate, 3, 25, 28–30
 - inquiring, 31
 - registration, 17, 23
 - resize, 3, 24
 - stride, 5, 17, 24, 34, 35
- assumption, 28
- attribute, 3–8, 14, 16, 24, 26, 28, 33–35
 - aggregate, 5, 6
 - bounds, 3, 18
 - checking, 33
 - hard, 17
 - inquiring, 32
 - setting, 17
 - soft, 17
 - vector components, 18
 - violation, 33
 - elemental, 5
 - existence, 21
 - initialization, 9, 16
 - inquiring, 29
 - keyword, 3, 14, 21, 24, 28
 - location, 14
 - mesh, 5, 7, 14, 21
 - nodal, 16
 - pane, 5, 14
 - parallel mesh, 21
 - redefine, 3, 14
 - registration
 - function pointer, 17
 - reinitiaize, 13
 - size, 3
 - registration, 16
 - unit, 14, 30
 - user-defined, 6, 13, 14, 16, 26
 - window, 6, 14, 16, 17, 19
- boundary condition, 5
- case sensitive, 10
- cells, 5
- CGNS, 16
- COM_abort, 12
- COM_allocate_array, 24, 25, 28
- COM_allocate_attribute (obsolete), 36
- COM_append_array, 24, 25
- COM_call_atexit (Fortran), 23
- COM_call_exit (Fortran), 23
- COM_call_function, 21
- COM_call_system (Fortran), 23
- COM_check_bounds, 33
- COM_clone_attribute, 27
- COM_clone_mesh (obsolete), 35
- COM_clone_mesh_sub (obsolete), 35
- COM_copy_array, 32
- COM_copy_attribute, 27
- COM_create_window (obsolete), 36
- COM_deallocate_array, 25
- COM_delete_attribute, 28
- COM_delete_pane, 28
- COM_delete_window, 28
- COM_EXTERN_MODULE (C/C++), 12
- COM_finalize, 11
- COM_free_buffer, 29–31
- COM_Func_ptr, 18
- COM_get_array, 31, 32
- COM_get_array_const, 3, 17, 31, 32
- COM_get_attribute, 30, 36
- COM_get_attribute_addr (obsolete), 36
- COM_get_attribute_handle, 21
- COM_get_attribute_handle_const, 21
- COM_get_attribute_info (obsolete), 36
- COM_get_attributes, 3, 29
- COM_get_bounds, 32
- COM_get_communicator, 13, 29

COM_get_connectivities, 3, 29
 COM_get_error_code, 10
 COM_get_function_handle, 21
 COM_get_object, 19, 26
 COM_get_panes, 3, 29, 36
 COM_get_parent, 31
 COM_get_pointer, 25, 26
 COM_get_size, 31, 36
 COM_get_status, 30
 COM_get_window_handle, 21
 COM_get_window_nelems (obsolete), 36
 COM_get_window_nnodes (obsolete), 36
 COM_get_window_npanes (obsolete), 36
 COM_get_window_panes (obsolete), 36
 COM_init, 11, 12
 COM_init_attribute (obsolete), 35, 36
 COM_init_attribute_strided (obsolete), 35
 COM_init_function (obsolete), 36
 COM_init_member_function (obsolete), 36
 COM_init_mesh (obsolete), 34, 35
 COM_init_profiling, 23
 COM_init_profiling (obsolete), 36
 COM_init_profiling_barrier (obsolete), 36
 COM_initialized, 11, 12
 COM_load_module, 12
 COM_LOAD_MODULE_... (C/C++), 12
 COM_Member_func_ptr, 19
 COM_new_attribute, 14, 34
 COM_new_window, 13, 36
 COM_Object, 19
 COM_print_profile, 22
 COM_resize_array, 24, 28
 COM_set_array, 3, 16, 17, 24
 COM_set_array_const, 3, 16, 17, 32
 COM_set_bounds, 17
 COM_set_external, 17
 COM_set_external_const, 17
 COM_set_function, 18, 36
 COM_set_member_function, 18, 19, 36
 COM_set_object, 19, 26
 COM_set_pointer, 25, 26
 COM_set_profiling, 22, 36
 COM_set_profiling_barrier, 22, 36
 COM_set_size, 3, 16, 17, 24
 COM_set_verbose, 11, 22
 COM_Type, 14, 18, 19, 30
 COM_unload_module, 12
 COM_UNLOAD_MODULE_... (C/C++), 12
 COM_use_attribute, 26, 27
 COM_use_mesh (obsolete), 35
 COM_use_mesh_sub (obsolete), 35
 COM_window_init_done, 13, 14, 28
 command-line, 9
 command-line options
 -com-home, 11
 -com-mpi, 11
 -com-v, 11
 compiler, 6, 18, 25, 33
 component, 4
 component-based, 4
 constant, 14
 coupling, 2
 CSAR, 1

 data integrity, 3, 7, 8, 17
 authorization, 4
 coherence, 7
 dangling, 7
 immutable, 3, 5, 7, 8, 21
 mutable, 5, 8, 21
 persistency, 4, 7, 8
 read-only, 17
 data type
 COM_BYTE, 37
 COM_CHAR, 37
 COM_CHARACTER, 37
 COM_COMPLEX, 37
 COM_DOUBLE, 37
 COM_DOUBLE_COMPLEX, 37
 COM_DOUBLE_PRECISION, 37
 COM_F90POINTER, 25, 37
 COM_FLOAT, 37
 COM_INT, 18, 19, 37
 COM_INTEGER, 18, 37
 COM_LOGICAL, 37
 COM_LONG, 37
 COM_LONG_LONG_INT, 37
 COM_METADATA, 19, 37
 COM_RAWDATA, 19, 37
 COM_REAL, 37
 COM_SHORT, 37
 COM_STRING, 19, 37
 COM_UNSIGNED, 37
 COM_UNSIGNED_CHAR, 37
 COM_UNSIGNED_LONG, 37
 COM_UNSIGNED_SHORT, 37

- COM_VOID, 25, 37
- directory, 33, 34
- element type, 15
 - bar, 15
 - brick, 15
 - pentahedron, 15
 - prism, 15
 - pyramid, 15
 - quadrilateral, 15
 - tetrahedron, 15
 - triangle, 15
 - wedge, 15
- environment variable
 - ROCCOM_HOME, 11
 - ROCSTAR_HOME, 11
- error handling, 10
- example, 5, 20
- external, 17–19, 25
- field, 5
- framework, 1, 2, 4, 8, 9, 12
- function, 4
 - existence, 21
 - initialization, 9
 - invocation, 6, 8, 9, 21
 - member function, 3, 6, 8, 18, 19
 - registration, 6, 18, 19
- functionality, 34
- handle, 3, 4, 8
 - attribute, 14, 21, 28
 - function, 21, 28
- header file
 - roccom.h, 10
 - roccomf90.h, 10
- incompatibility, 6
- inheritance, 3, 4, 6, 7, 14, 24, 26–28, 34
 - clone, 7, 26
 - copy, 7, 25
 - duplicate, 26, 27, 34
 - pane connectivity, 15
 - subwindow, 6
- initialization, 3, 11
- input, 6, 18
- integration, 1, 4
- inter-module, 4, 6, 9
- interaction, 10
- interface, 4, 6, 8, 10, 18, 36
- interoperability, 1, 8, 19, 37
- library, 3, 33
 - dynamic, 12
 - name, 12
- limitation, 19
- makefile, 33, 34
 - dependency, 33
 - gmake, 34
- memory
 - allocate, 36
 - contiguous, 5
 - debugging, 9
 - F90 pointer, 25
 - free, 25
 - malloc, 9
 - staggered, 5, 34
 - user-allocated, 23, 24
 - user-registered, 8
- mesh
 - connectivity, 3, 5, 8, 14–17, 21, 27, 30, 35, 36
 - :B20, 15
 - :B8, 15
 - :H8, 15
 - :P14, 15
 - :P15, 15
 - :P18, 15
 - :P5, 15
 - :P6, 15
 - :T10, 15
 - :T4, 15
 - :b2, 15
 - :q4, 15
 - :q8, 15
 - :q9, 15
 - :st1, 15
 - :st2, 15
 - :st3, 15, 35
 - :t3, 15
 - :t6, 15
 - inquiring, 30
 - registration, 16
- coordinates, 5, 14, 21, 34
- dimension, 5
- ghost, 5, 16, 27, 31
- mixed, 3, 5, 8

- multi-block, 5, 23
- pane connectivity, 5, 14, 21
 - format, 15
- structured, 5, 15, 16, 31, 35, 36
 - dimensions, 15, 31
- unstructured, 5
- migrating, 34
- mixed-language, 6
- mixing, 33
- module
 - loading, 3, 12
 - unload, 3
- module
 - categorization, 1
 - finalization, 11
 - initialization, 11
 - load, 9
 - requirements, 9
 - unloading, 12
- MPI, 2, 13, 29
 - communicator, 13, 22, 29
 - MPI_Barrier, 22
 - MPI_Comm, 13, 22, 29
- multi-thread, 2
- object oriented, 4, 6, 18, 26
 - class, 8
 - data member, 13
 - encapsulation, 4
 - function, 13
 - polymorphism, 4
 - runtime, 6
 - public, 2, 4, 13
- pane, 2, 4, 5, 8, 16
 - deletion, 28
 - ID, 5, 17
 - inquiring, 29
- pane connectivity, 14
 - inheritance, 15
- PAPI, 9
- parallel, 1, 2, 4–6, 22
- platform-dependent, 33
- plug-and-play, 2
- predictor-corrector, 9
- profiling, 4, 9, 22
 - barrier, 22
 - elapsed, 23
 - output, 22
 - wall-clock, 23
- registration, 2–7, 13, 14, 16, 17, 19, 34
- reinitialization, 4
- restart, 9
- Rocman, 2, 9, 23, 26
- Rocstar, 1, 2, 4, 9
- runtime, 4, 8
- runtime system, 1, 2, 8, 9, 11
- scalar, 17
- shutdown, 11
- STATIC_LINK, 13
- string
 - null termination, 8, 10, 19
- tracing, 4, 9, 11, 22
- two-dimensional, 5
- uppercase, 19
- window, 3, 4, 6–8, 13, 34
 - creation, 13
 - data member, 4
 - deletion, 9, 28
 - existence, 21
 - initialization, 9
 - inquiring, 6, 8, 28
 - split, 26