Engineering with Computers
DOI 10.1007/s00366-006-0034-x

ORIGINAL ARTICLE

A system integration framework for coupled multiphysics simulations

Xiangmin Jiao · Gengbin Zheng · Phillip A. Alexander · Michael T. Campbell · Orion S. Lawlor · John Norris · Andreas Haselbacher · Michael T. Heath

Received: 19 April 2005 / Accepted: 1 February 2006
© Springer-Verlag London Limited 2006

Abstract Multiphysics simulations are playing an increasingly important role in computational science and engineering for applications ranging from aircraft design to medical treatments. These simulations require integration of techniques and tools from multiple disciplines, and in turn demand new advanced technologies to integrate independently developed physics solvers effectively. In this paper, we describe some numerical, geometrical, and system software components required by such integration, with a concrete case study of detailed, three-dimensional, parallel rocket simulations involving system-level interactions among fluid, solid, and combustion, as well as subsystem-level interactions. We package these components into a software framework that provides common-refinement based methods for transferring data between potentially non-matching meshes, novel and robust face-offsetting methods for tracking Lagrangian surface meshes, as well as integrated support for parallel mesh optimization, remeshing, algebraic manipulations, performance monitoring, and high-level data management and I/O. From these general, reusable framework components we construct domain-specific building blocks to facilitate integration of parallel, multiphysics simulations from high-level specifications that are easy to read and can also be visualized graphically. These reusable building blocks are integrated with independently developed physics codes to perform various multiphysics simulations.

Keywords Software framework · Multiphysics simulation · System integration · Data abstraction

X. Jiao · G. Zheng · P. A. Alexander · M. T. Campbell · O. S. Lawlor · J. Norris · A. Haselbacher · M. T. Heath
Center for Simulation of Advanced Rockets,
University of Illinois, Urbana, IL 61801, USA

X. Jiao (✉)
College of Computing, Georgia Institute of Technology,
Atlanta, GA 30332, USA
e-mail: jiao@cc.gatech.edu

Present Address:
O. S. Lawlor
Department of Computer Science, University of Alaska,
Fairbanks, AK, USA

1 Introduction

Many real-world systems involve complex interactions between multiple physical components. Examples include natural systems, such as climate models, as well as engineered systems, such as automobile, aircraft, or rocket engines. Simulation of such systems helps improve our understanding of their function or design, and potentially leads to substantial savings in time, money, and energy.

Simulation of multicomponent systems poses significant challenges in the physical disciplines involved, as well as computational mathematics and software systems. In terms of software design, the data exchanged between modules must be abstracted appropriately so that inter-module interfaces can be as simple and clean as possible. The software architecture must encourage good software practice, such as encapsulation and code reuse, and provide convenience to code developers while being non-intrusive. In addition, the framework must provide computational services to allow sufficient flexibility for application scientists and engineers to choose appropriate

discretization schemes, data structures, and programming languages according to their tastes and needs. Finally, to support cutting-edge research, the software architecture must maximize concurrency in code development of different subgroups and support rapid prototyping of various coupling schemes through well-defined service components. In recent years, several software frameworks have been developed for large-scale scientific applications, such as Cactus [1], CCA [2], Alegra [3], Overture [4], POOMA [5], and Sierra [6]. These frameworks share some similar objectives and address different aspects of these challenges, but domain-specific high-level software frameworks are still needed for coupled multiphysics simulations such as fluid–solid interactions.

In this paper we describe the software framework developed at the Center for Simulation of Advanced Rockets (CSAR) at the University of Illinois for large-scale integrated rocket simulations. Our framework provides a higher-level object-oriented abstraction of interface data and functions to enable clean and simple inter-module interfaces. On top of the abstraction, our framework provides a set of service components fine-tuned for quick integration of multiphysics simulations. We provide a technical overview of the computational and computer science support for these rocket simulations.

The remainder of the paper is organized as follows. Section 2 briefly overviews the motivating application of our integration framework and its software components. Section 3 presents the communication-oriented integration interface for multicomponent systems. Section 4 describes a few service utilities to support integration of such systems. Section 5 describes a high-level orchestration framework for the integrated rocket simulations. Section 6 shows some performance results of couple simulations using our framework. Section 7 concludes the paper.

2 Motivating application

The motivating multiphysics application for the integration framework described in this paper is an ongoing project at CSAR. The ultimate objective of CSAR is to develop an integrated software system, *Rocstar*, for detailed whole-system simulation of solid rocket motors under normal and abnormal operating conditions. This software system is applicable, however, to systems beyond rockets, such as simulations of gas turbines, flapping wings, and arterial blood flows. We briefly overview the methodology and the software components of this system.

2.1 Coupling methodology

Simulation of a rocket motor involves many disciplines, including three broad physical disciplines—fluid dynamics, solid mechanics, and combustion—that interact with each other at the primary system level, with additional subsystem-level interactions, such as particles and turbulence within fluids. Because of its complex and cross-disciplinary nature, the development of *Rocstar* has been intrinsically demanding, requiring diverse backgrounds within the research team. In addition, the capabilities required from the individual physical disciplines are at the frontier of their respective research agendas, which entails rapid and independent evolution of their software implementations.

To accommodate the diverse and dynamically changing needs of individual physics disciplines, we have adopted a *partitioned* approach to enable coupling of individual software components that solve problems in their own physical and geometrical domains. With this approach, the physical components of the system are naturally mapped onto various software components (or modules), which can then be developed and parallelized independently. These modules are then integrated into a coherent system through an integration framework, which, among other responsibilities, manages the distributed meshes and associated attributes for finite element or finite volume methods and performs inter-module communications on parallel machines.

2.2 Software architecture

To enable parallel simulations of rockets, we have developed a large number of software modules. Figure 1 shows an overview of the components of the current generation of *Rocstar*. These modules serve very diverse purposes and have diverse needs in their

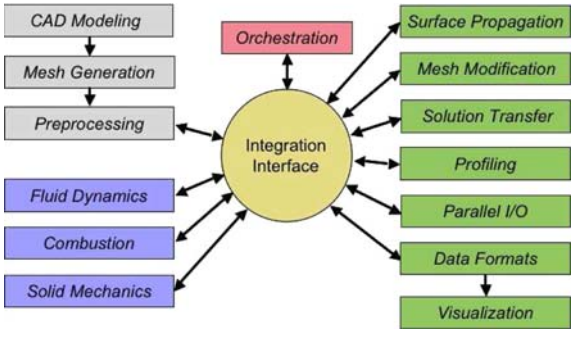


Fig. 1 Overview of *Rocstar* software

development and integration. We loosely group these modules into the following four categories.

Physics modules solve physical problems in their respective geometric domains. In general, they are similar to stand-alone applications, are typically written in Fortran 90 (F90), and use array-based data structures encapsulated in derived data types.

Integration interface provides data management and function invocation mechanisms for inter-module interactions.

Service modules provide specific service utilities, such as I/O, communication, and data transfer. They are typically developed by computer scientists but driven by the needs of applications, and are usually written in C++.

Orchestration (*control*) *modules* specify overall coupling schemes. They contain high-level, domain-specific constructs built on top of service modules, provide callback routines for physics modules to obtain boundary conditions, and mediate initialization, execution, finalization, and I/O for physics and service modules.

In *Rocstar*, the above categories correspond to the components at the lower-left, center, right, and top, respectively, of Fig. 1. In addition, our system uses some *off-line tools*, such as those in the upper-left corner of Fig. 1, which provide specific pre- or post-processing utilities for physics modules. The focus of this paper is the last three categories, which compose a hierarchical framework. In the following sections, we describe these software components in more detail.

3 Integration interface

To facilitate interactions between modules, we have developed an object-oriented, data-centric framework called *Roccom*. *Roccom* utilizes an object-oriented methodology for abstracting and managing the data and functions of a module. This abstraction is mesh- and physics-aware and programming-language neutral, and supports encapsulation, polymorphism, and inheritance. *Roccom* simplifies inter-module interactions through high-level abstractions, and allows the individual components to be developed as independently as possible and integrated subsequently with little or no changes.

3.1 Data management

3.1.1 Object-oriented abstraction

Roccom organizes data into distributed objects called *windows*. A window encapsulates a number of *data attributes* (such as the mesh and some associated field variables) 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 attributes, although the sizes of attributes may vary. A module constructs windows at runtime by creating attributes and registering the addresses of the attributes and functions. In *Rocstar*, each physics module typically has a volume window to encapsulate the volumetric data and a surface window to encapsulate the boundary of the volume mesh. The surface patches with different boundary conditions are mapped to different panes to simplify treatment of boundary conditions. The attributes registered with *Roccom* are typically *persistent* (instead of temporary) datasets, in the sense that they last beyond a major coupled simulation step. Different modules can communicate with each other only through windows, as illustrated in Fig. 2.

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 *Roccom* 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.

3.1.2 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 window or pane attribute is associated

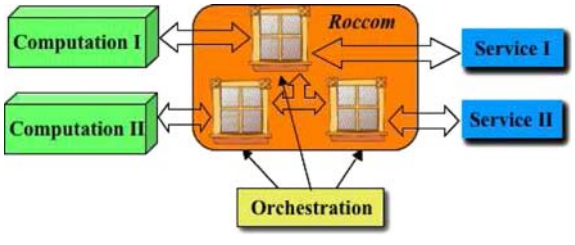


Fig. 2 Schematic of windows and panes

with a window or pane (such as some control parameters and boundary-condition flags), respectively.

Attribute layout Within a pane, an attribute is conceptually a two-dimensional dataset: one dimension corresponds to the items (such as nodes or elements for nodal and elemental attributes, respectively), and the other dimension corresponds to the components (such as *x*-, *y*-, *z*-components) per item. The data storage can be in a *pointwise*, *block*, or *strided* layout. In the pointwise layout, the attribute is stored in an array in which the components associated with each item (node or element) are stored contiguously. In the block layout, each component corresponding to different items are stored contiguously, and the attribute within the pane may be stored contiguously or in separate arrays. In the strided layout, there is a constant stride for each component between two adjacent items. The strided layout is more general in that the pointwise and block layouts are its special cases. Figure 3 illustrates these different layouts.

Mesh data In general, the name and type of an attribute are defined by users, with the exception of mesh data. Mesh data include nodal coordinates and element connectivity, whose attribute names and data types are predefined by *Roccom*. The nodal coordinates (''nc'') are double-precision floating-point numbers, with three components per node. If the coordinates of a pane are stored contiguously, the storage can be registered using the name ''nc''; otherwise, the *x*-, *y*-, and *z*- components must be registered separately using names ''1-nc'', ''2-nc'', and ''3-nc'', respectively.

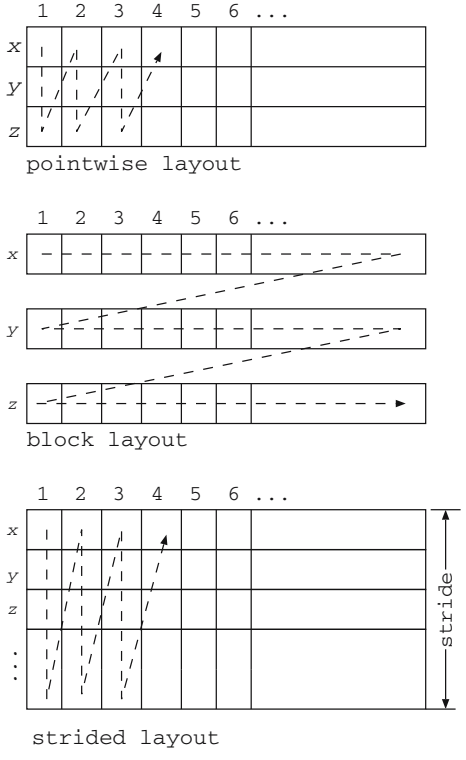


Fig. 3 Different data layouts supported by *Roccom*

In *Roccom*, element connectivity is not a regular attribute, because different panes can have different element types. To differentiate a connectivity table from a regular attribute, the name of a connectivity table has two parts: the first part indicates the type of the element, in the format of a colon followed by a type ID (e.g., ''t3'' or ''q4'' for 3-node triangles and 4-node quadrilaterals, respectively); the second part is a user-defined name to distinguish different connectivity tables of the same element type, and is separated from the first part by a colon (e.g., ''t3:ghost'').

Roccom 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. Structured meshes have no connectivity tables, and the shape of a pane is registered using the name ''st''. For unstructured meshes, each pane has one or more connectivity tables, where each connectivity table contains consecutively numbered elements of the same type. Each connectivity table must be stored in an array with pointwise or block layout.

To facilitate parallel simulations, *Roccom* 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. In addition, each pane can have a *pane connectivity*, which contains the communication information for shared nodes along partition boundaries and for ghost nodes and ghost elements in a predefined format.

Aggregate attributes In *Roccom*, 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 connectivities; the name ''all'' refers to all the data attributes in a window. One can use ''*i*-attribute'' ($i \geq 1$) to refer to the *i*th component of each attribute or use ''attribute'' to refer to all the components collectively.

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 to have *Roccom* allocate memory for all the defined but unregistered attributes.

3.2 Inheritance

Roccom also introduces the novel concept of *partial inheritance* of windows to construct a subwindow by *using* or *cloning* a subset of the mesh or attributes of another window. 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 4 depicts a scenario of inheritance among three windows.

To support these needs, *Roccom* allows inheriting the mesh from a parent window to a child window in either of two modes. First, inherit the mesh of the parent as a whole. Second, inherit only a subset of panes that satisfy a certain criterion, with the option to exclude the ghost nodes and cells of the parent from the child. 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 *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, *Roccom* overwrites the existing attribute with the new attribute.

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

3.3 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. We address these issues through the mechanisms of persistency and immutable references, respectively.

Persistency *Roccom* maintains references to the datasets registered with its windows. To avoid dangling references associated with data registration, we impose the following persistency requirement: the datasets registered with a window must outlive the life of the window. This notion of persistency is simple and intuitive, and is sometimes used as the ''preferred approach to implementing systems'' in similar contexts such as object-oriented databases [7]. 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 system; instead, we treat it as a design pattern that application code developers should follow. Fortunately, typical physics codes allocate memory spaces during an initialization stage and deallocate memory during a finalization stage, which naturally adapts to this design pattern.

Immutable references Another potential issue for data integrity is side effects due to inadvertent changes to datasets. To address this problem, some traditional object-oriented paradigms require that a client can change the state of a supplier object only through the supplier's public interfaces. However, it has been noted that this integrity model is unnecessarily restrictive for complex systems [8]. For the internal states of the modules, *Roccom* facilitates the traditional integrity model through member functions that we will describe shortly. For interface datasets, we enforce access control for immutable handles of data attributes. In *Roccom*, a service module can obtain access 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. Furthermore, as we describe later, service modules of *Roccom* are implemented using a C++ interface that conforms to immutable references at the language level, so *Roccom*'s application can be free of side effects with minimal runtime overhead.

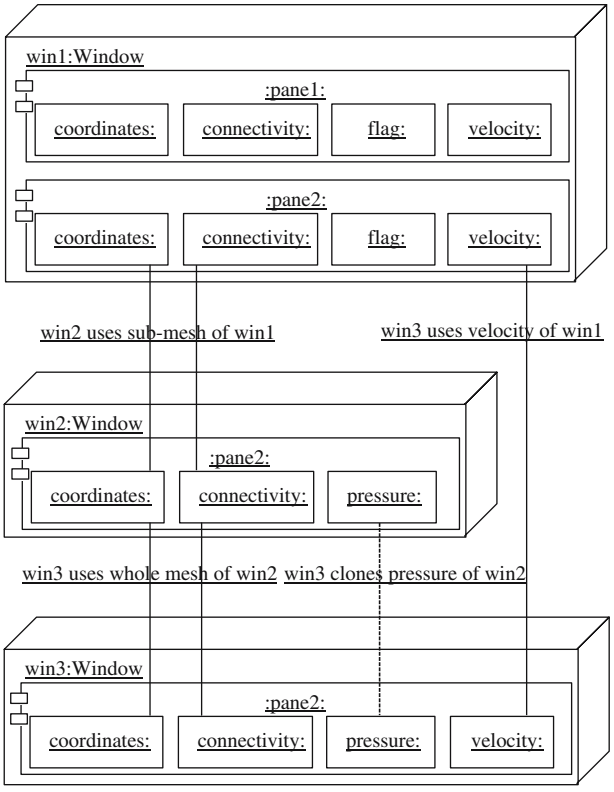


Fig. 4 Scenario of inheritance of mesh and field attributes among three windows

3.4 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 languages, 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. In addition, the registered functions can be regular C++ member functions or even virtual functions. This feature allows advanced language interoperability between C++ and F90 without sacrificing object-orientedness of the interface of complex modules.

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.

3.5 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.

3.5.1 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 some other frameworks (such as the BABEL of CCA [2]), except that *Roccom* parses the definitions of the windows at runtime. *Roccom* provides different bindings for C++ and F90, with similar semantics. In the following, we mention a few highlights of the API.

Data management and retrieval The basic interface functions that all modules must use are the construction of windows and registration of data attributes. Figure 5 shows a sample F90 code fragment that creates a window with two panes. Typically, data registered in a window are accessed by service modules through C++ interfaces, which can enforce data integrity as discussed later. A physics module may also want to access a dataset through *Roccom*, for example, if a window was created by, or inherited from, another

```
INTEGER           ::  nn1, nn2    ! sizes of nodes
INTEGER           ::  ne1, ne2    ! sizes of elements
INTEGER           ::  types(2)
INTEGER, POINTER          ::  conn1(3,ne1), conn2(4,ne2)
DOUBLE PRECISION, POINTER  ::  coors1(3,nn1), coor2(3,nn2)
DOUBLE PRECISION, POINTER  ::  disp1(3,nn1), disp2(3,nn2)
DOUBLE PRECISION, POINTER  ::  loc1(ne1,3), loc2(ne2,3)
EXTERNAL   fluid_update

CALL COM_NEW_WINDOW("fluid")

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

! Create an element-centered double-precision dataset
CALL COM_NEW_ATTRIBUTE("fluid.loc", "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)
CALL COM_SET_SIZE("fluid.:t3:", 11, ne1)
CALL COM_SET_ARRAY("fluid.:t3:", 11, conn1)

! Create a pane with ID 21 of a quadrilateral surface mesh
CALL COM_SET_SIZE("fluid.nc", 21, nn2)
CALL COM_SET_ARRAY("fluid.nc", 21, coors2)
CALL COM_SET_SIZE("fluid.:q4", 21, ne2)
CALL COM_SET_ARRAY("fluid.:q4", 21, ne2)

! Register addresses of data attributes in block layout
CALL COM_SET_ARRAY("fluid.disp", 11, disp1, 1)
CALL COM_SET_ARRAY("fluid.loc", 11, loc1, 1)
CALL COM_SET_ARRAY("fluid.disp", 21, disp2, 1)
CALL COM_SET_ARRAY("fluid.loc", 21, loc2, 1)

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

CALL COM_WINDOW_INIT_DONE("fluid")
```

Fig. 5 Sample F90 code fragment for window registration

module. To support this need, *Roccom* provides an API for retrieving information about panes and attributes, such as the number of panes, the list of pane IDs, the numbers of nodes and elements in the panes, and the metadata of attributes. As an advanced feature, *Roccom* allows an F90 code to obtain the addresses of a dataset in *Roccom* through F90 pointers, which would then make the F90 code assume ownership of the dataset. This feature enables the capability of managing memory spaces in C++ for F90 codes, which is convenient for developing some service utilities. Because ownership is transferred to the F90 code, data integrity is not compromised.

Function registration and invocation A module registers a function with *Roccom* in a similar manner to registering window attributes. The arguments of a registered function can be pointers or references to primitive data types (such as integer, double, or char), or, more powerfully, pointers to Attribute objects (typically for service utilities) or to the raw address registered with a window attribute (as with the context object of a member function). To register a function, a module specifies a function pointer and the number, intentions (for input or output), and base data types of its arguments. For technical reasons, we impose a limit on the maximum number of the arguments that a registered function can take, but the limit can be adjusted, if desired, by a minor change to *Roccom*'s implementation.

Inter-module function invocation is done through *Roccom*, as demonstrated in Fig. 6. COM_call_function takes the handle of the callee function, the number of arguments, and the actual arguments to be passed to the callee. If an argument of the callee is an Attribute object, the caller passes a reference to the handle of the attribute. This allows mixed-language interoperability. For data integrity, *Roccom* enforces that an immutable handle can be passed only to a read-only input argument. In a parallel setting, the invoked function will typically be executed on the same processor as the caller, supporting SPMD style parallelism.

Dynamic loading of modules In the *Roccom* framework, each module can be built into a shared object,

```
INTEGER           ::  f_h               ! function handle
INTEGER           ::  a1_h, a2_h, a3_h  ! attribute handles

f_h  = COM_GET_FUNCTION_HANDLE("Rocblas.add")
a1_h = COM_GET_ATTRIBUTE_HANDLE_CONST("fluid.nc")
a2_h = COM_GET_ATTRIBUTE_HANDLE_CONST("fluid.disp")
a3_h = COM_GET_ATTRIBUTE_HANDLE("fluid.loc")

! Compute loc=nc+disp
CALL COM_CALL_FUNCTION(f_h, 3, a1_h, a2_h, a3_h)
```

Fig. 6 Sample F90 code fragment for function invocation

which is linked into the executable only at runtime. A dynamically loaded shared object facilitates plug-and-play of modules, and can also effectively avoid name-space pollution among modules, because such an object can have its own local name scope. *Roccom* accommodates dynamic loading by providing a COM_load_module function, which takes a module's name and a window name as arguments, and loads the shared object of the module using the dynamic linking loader dlopen. Each module provides an initialization routine *Module*_load_module, which constructs a window with a given name. *Roccom* tries to locate the routine using both the C and Fortran naming conventions and then invokes it following the corresponding calling convention. This technique further enhances transparency of C++/F90 interoperability.

3.5.2 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 Fig. 7, on a UML class diagram [9].

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 immutable references so that a function can navigate through only immutable references if the root reference was immutable. Through this C++ interface developers implement service utilities (such as transferring data across different meshes) independently of application codes.

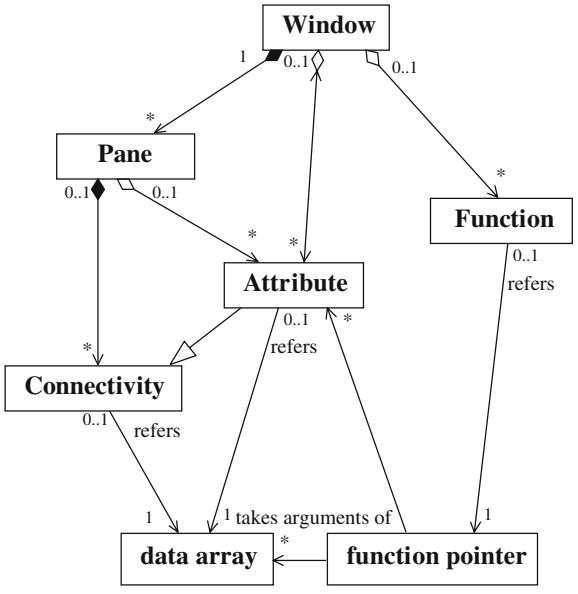


Fig. 7 UML associations of *Roccom*'s classes

3.5.3 Roccom runtime system

The runtime system 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 look-up, 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 translator 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.

Through the calling mechanism, *Roccom* also provides tracing and profiling capabilities for inter-module calls to aid in debugging and performance tuning. It also exploits hardware counters through PAPI [10] 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. For submodule-level profiling, profiling services are provided through the standard MPI_Pcontrol interface, as well as a native interface for non-MPI based codes. By utilizing the MPI_Pcontrol interface, applications developers can collect profiling information for arbitrary, user-defined sections of source code without breaking their stand-alone codes.

3.6 Message passing communication subsystem

In the *Rocstar* code suite, each of its physics components—fluids, solids, and combustion—began as an independently developed parallel message passing program written using MPI to maximize portability. These rocket simulations involve dynamically changing geometry, and hence may require mesh adaptivity and dynamic load balancing. Typical implementations of MPI offer little or no automatic support for such dynamic behaviors. As a result, programming productivity and parallel efficiency may suffer.

Adaptive MPI (AMPI) [11, 12] is an adaptive and portable implementation of MPI that exploits the idea

of *processor virtualization* [13] to tackle this challenge. AMPI, while still retaining the familiar programming model of MPI, is better suited for such complex applications with a dynamic nature. AMPI and its underlying system CHARM++ are developed at the Parallel Programming Laboratory led by Professor Kalé at University of Illinois at Urbana-Champaign in collaboration with CSAR. The *Roccom* system provides integrated support to ease adapting its software components to take advantage of processor virtualization.

3.6.1 Processor virtualization

The key concept behind AMPI is processor virtualization. Standard MPI programs divide the computation into P processes, and typical MPI implementations simply execute each process on one of the P processors. In contrast, an AMPI programmer divides the computation into a number V of virtual processors (VPs), and the AMPI runtime system maps these VPs onto P physical processors. In other words, AMPI provides an effective division of labor between the programmer and the system. The programmer still programs each process with the same syntax as specified in the MPI Standard. Further, not being restricted by the physical processors, the programmer is able to design more flexible partitioning that best fits the nature of the parallel problem. The runtime system, on the other hand, has the opportunity of adaptively mapping and re-mapping the programmer's virtual processors onto the physical machine. Adaptive MPI implements its MPI processors as CHARM++ user-level threads bound to CHARM++ communicating objects (See Fig. 8). During execution, several MPI ''processors'' can run on one physical processor as user-level threads.

3.6.2 Integration with AMPI

In the AMPI execution environment, several MPI threads run in one process. Thus, global variables in the application must be privatized so that each MPI thread has access to its own copies of global variables. One simple solution adopted in *Rocstar* is to collect all global variables of each module into a global data structure, which is then passed as a parameter to each function that requires access to the global variables. This global structure is allocated per thread at the initialization phase, and is registered with *Roccom* as an attribute associated with the window of that module. This attribute is then designated as the context object of the ''member functions'' of the window, and is passed to the function implicitly at runtime by *Roccom*. Each thread has a private copy of *Roccom*, contained in an array of *Roccom* objects. *Roccom* and CHARM++ have been prewired so that the proper *Roccom* object is selected during a context switch. This approach allows the application components to take advantage of processor virtualization with little effort, and at the same time encourages object-oriented design of the components.

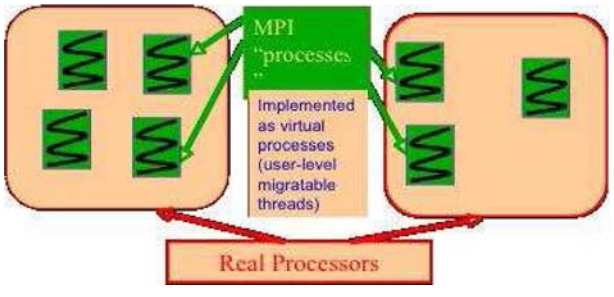


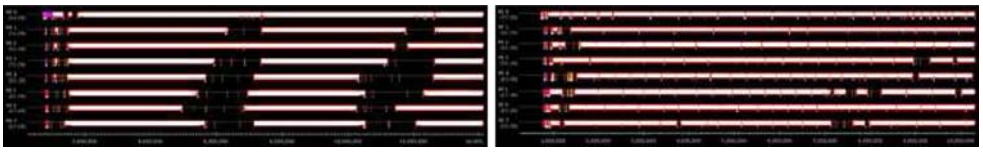
Fig. 8 Implementation of AMPI virtual processors

3.6.3 Benefits of processor virtualization

The benefits of processor virtualization in parallel programming are discussed in detail in [13]. The CHARM++ system takes full advantage of these benefits. AMPI inherits most of the merits from CHARM++, while furnishing the familiar MPI programming environment. The following is a list of the benefits enjoyed by the AMPI-enabled *Rocstar*.

Automatic load balancing If some of the physical processors become overloaded, the runtime system can migrate a few of their MPI threads to relatively underloaded physical processors. The AMPI runtime system and load balancing framework [14] can make such load balancing decision based on automatic instrumentation.

Adaptive overlapping of communication and computation If one of the MPI threads is blocked on a receive, another MPI thread on the same physical processor can run. This largely eliminates the need for the programmer to specify manually some static computation/communication overlapping, as is often required in MPI. Figure 9 illustrates an example using the *Projections* [15] visualization tool. The solid blocks represent computation and the gaps are idle time when CPU is waiting for incoming messages. As the degree of virtualization (number of MPI threads on each physical processor) increases, there are more opportunities for the smaller blocks to fill in the gaps (idle time) and consequently the CPU utilization increases.



(a) Profiles of 8 threads on 8 processors. **(b)** Profiles of 64 threads on 8 processors.

Fig. 9 Adaptive overlapping of communication and computation

Flexibility to run on arbitrary number of processors Since more than one MPI threads can be executed on one physical processor, AMPI is capable of running MPI programs on any arbitrary number of processors. This feature proves to be useful in application development and debugging phases. This is one of the most notable benefits of AMPI that CSAR developers cherish. For example, one may face a communication bug that manifests itself only when the dataset was partitioned for 480 processors. Finding and fixing the problem would be very difficult, as such a large number of processes are hardly available in debugging or interactive mode and may require long waiting time to obtain even in batch mode at supercomputer centers. Using AMPI, the developers are able to debug the problem interactively, using 480 MPI threads distributed over a small number of physical processors on a local cluster, resolving the problem in a more productive manner.

Processor virtualization potentially may lead to extra overhead due to the thread scheduling cost and more fine grained communication. It is thus a trade-off to select the degree of virtualization independent of the number of processors. We have demonstrated that virtualization has minimal performance penalty [16] in terms of the thread scheduling overhead, due to the efficient implementation of thread context switching [17]. In order to justify a lower granularity (hence a high degree of virtualization), the amount of computation associated with each message must be substantially larger than the per message overhead (typically around 10 μs per message and a couple of nanoseconds per byte). Other factors influencing the decision are cache effects. AMPI runtime system promotes better cache performance, which leads to improved performance. A virtual processor handles a smaller set of data than a physical processor, so a virtual processor will have better memory locality. This blocking effect is the same method many *serial* cache optimizations employ, and AMPI programs get this benefit automatically.

4 Framework service utilities

On top of *Roccom*, we have developed a number of reusable service modules, including *middleware* services, such as communication and I/O, as well as *computational* services, such as data transfer and mesh optimization. In the following, we describe these services and their roles in the integrated simulations.

4.1 Interpane communication

Traditional message-passing paradigms typically provide general but low-level inter-process communications, such as send, receive, and broadcast. In physical simulations using finite element or finite volume methods, communications are typically across panes or partitions, whether the panes or partitions are on the same or different processes. The *Roccom* framework provides high-level inter-pane communication abstractions, including performing reductions (such as sum, max, and min operations) on shared nodes, and updating values for ghost (i.e., locally cached copies of remote values of) nodes or elements. Communication patterns between these nodes and elements are encapsulated in the *pane connectivity* of a window, which can be provided by application modules or constructed automatically in parallel using geometric algorithms. These inter-pane communication abstractions simplify parallelization of a large number of modules, including surface propagation and mesh smoothing, which we will discuss shortly.

4.2 Data input/output

In scientific simulations, data exchange between a module and the outside world can be very complex. For file I/O alone, a developer must already face many issues, including various file formats, parallel efficiency, platform compatibility, and interoperability with offline tools. In a dynamic simulation, the situation is even more complex, as the code may need to exchange its mesh and data attributes with mesh repair or remeshing services, or receive data from remote processes.

To meet these challenges, we use the *window* abstraction of *Roccom* as the medium or ''virtual file'' for all data exchanges for a module, regardless whether the other side is a service utility, files of various formats, or remote machines, and let middleware services take care of the mapping between the window and the other side. For example, file I/O services map *Roccom* windows with scientific file formats (such as HDF and CGNS), so that the details of file formats and optimization techniques become transparent to application modules. Furthermore, as illustrated in Fig. 10, all application modules obtain data from an input window through a generic function interface, obtain_attribute(), which is supported by a number of services, including file readers and remeshing tools. This design allows physics modules to use the same initialization routine to obtain data under different circumstances, including initial startup, restart, restart after remeshing, and reinitialization after mesh repair.

4.3 Inter-mesh data transfer

In multiphysics simulations, the computational domains for each physical component are frequently meshed independently. This in turn requires geometric algorithms to correlate the surface meshes at the common interface between each pair of interacting domains to exchange boundary conditions. These surface meshes in general have different connectivities and may have gaps between them or interpenetrate each other. In general, they are also partitioned differently for parallel computation.

To correlate such disparate interface meshes, we have developed an efficient and robust algorithm to construct a *common refinement* of two triangular or quadrilateral meshes modeling the same surface [18, 19]. The common refinement is a finer mesh whose polygons subdivide the polygons of the input surface meshes. Our algorithm constructs a nearly orthogonal projection between two mesh surfaces that gives a continuous and one-to-one correspondence between their respective geometric realizations. Based on this projection, intersections of edges are defined and then used to compute the common refinement. The projections involve non-linear equations that can be solved only approximately by iteration, and the numerical errors can potentially cause topological inconsistencies. We achieve robustness through a combination of techniques, including error analysis and associated tolerancing, detection of inconsistencies, and automatic resolution of such inconsistencies using topological operations. For more detail, readers are refer to [18] and [19].

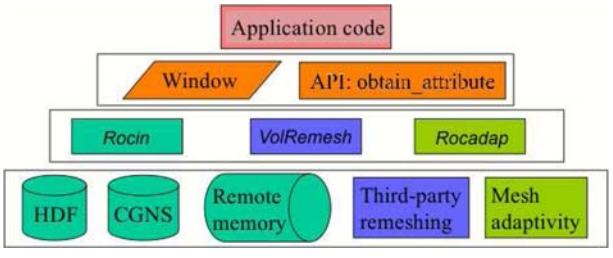


Fig. 10 Abstraction of data input

After constructing the common refinement, we must transfer data between the non-matching meshes in a numerically accurate and physically conservative manner. Some traditional methods, such as pointwise interpolation and some weighted residual methods [20], can achieve either accuracy or conservation but could not achieve both simultaneously. Our data transfer algorithm minimizes errors in the L_2 or Sobolev norm while achieving strict conservation, similar to the mortar element method for non-conforming domain decomposition [21, 22]. Leveraging the common refinement, our implementation achieves high accuracy and enforces conservation to nearly machine precision, significantly enhancing the accuracy of multiphysics simulations [23, 24]. For parallel runs, the common refinement also provides the correlation of elements across partitions of different meshes, and hence provides the communication structure needed for inter-module, inter-process data exchange.

4.4 Surface propagation

In *Rocstar*, the interface must be tracked as it regresses due to burning. In recent years, Eulerian methods, especially level set methods, have made significant advancements and become the dominant methods for moving interfaces [25, 26]. In our context, Lagrangian representation of the interface is crucial to describe the boundary of volume meshes of physical regions. However, previous numerical methods, either Eulerian or Lagrangian, have difficulties in capturing the evolving singularities (such as ridges and corners) in solid rocket motors.

To meet this challenge, we have developed a novel method, called *face-offsetting*, based on a new *entropy-satisfying Lagrangian formulation*. Face-offsetting methods deliver an accurate and stable entropy-satisfying solution without requiring Eulerian volume meshes. A fundamental difference between face-offsetting and traditional Lagrangian methods is that our methods solve the Lagrangian formulation face by face, and then reconstruct vertices by constrained minimization

and curvature-aware averaging, instead of directly moving vertices along some approximate normal directions. This method allows part of the surface to be fixed or to be constrained to move along certain directions (such as constraining the propellant to burn along the case). It supports both structured and unstructured meshes, with an integrated node redistribution scheme that suffices to control mesh quality for moderately moving interfaces. Figure 11 shows the propagation of a block-structured surface mesh for the fluids domain of the Attitude Control Motor (ACM) rocket, where the front and aft ends burn along the cylindrical case.

When coupled with mesh adaptation, the face-offsetting method can capture significant burns. Figure 12 shows a sample result of the burning of a star grain section of a rocket motor using the face-offsetting method coupled with surface remeshing using MeshSim from Simmetrix (http://www.simmetrix.com). The interior (the fins) of the propellant burns at uniform speed and exhibits rapid expansion at slots and contraction at fins. The fin tips transform into sharp ridges during propagation, as captured by the face-offsetting method.

4.5 Mesh optimization

In *Rocstar*, each physics module operates on some type of mesh. An outstanding issue in integrated rocket simulations is the degradation of mesh quality due to the changing geometry resulting from consumption of propellant by burning, which causes the solid region to shrink and the fluid region to expand, and compresses or inflates their respective meshes. This degradation can lead to excessively small time steps when an element becomes poorly shaped, or even outright failure when an element becomes inverted. Some simple mesh motion algorithms are built into our physics modules. For example, simple Laplacian smoothing is used for unstructured meshes, and a combination of linear transfinite interpolation (TFI) [27] with Laplacian smoothing is used for structured meshes in *Rocflo*. These simple schemes are insufficient when the meshes undergo major deformation or distortion. To address this issue, we take a three-tiered approach, in increasing order of aggressiveness: mesh smoothing, mesh repair, and global remeshing.

Mesh smoothing copes with gradual changes in the mesh. We provide a combination of in-house tools and integration of external packages. Our in-house effort focuses on parallel, feature-aware surface mesh optimization, and provides novel parallel algorithms for mixed meshes with both triangles and quadrilaterals. To smooth volume meshes, we utilize the serial MESQUITE package [28] from Sandia National Laboratories, which also works for mixed meshes, and we parallelized it by leveraging our across-pane communication abstractions.

If the mesh deforms more substantially, then mesh smoothing becomes inadequate and more aggressive mesh repair or even global remeshing may be required, although the latter is too expensive to perform very frequently. For these more drastic measures, we currently focus on only tetrahedral meshes, and leverage third-party tools off-line, including Yams and TetMesh from Simulog and MeshSim from Simmetrix, and we have work in progress to integrate MeshSim into our framework for online use. Remeshing requires that data be mapped from the old mesh onto the new mesh, for which we have developed parallel algorithms to transfer both node- and cell-centered data accurately, built on top of the parallel collision detection package developed by Lawlor and Kalé [29]. Figure 13 shows an example where the deformed star grain is remeshed with the temperature field of the fluids volume transferred from the old to the new mesh.

5 Orchestration framework

In coupled rocket simulations, the individual physics modules solve for the solutions on their respective

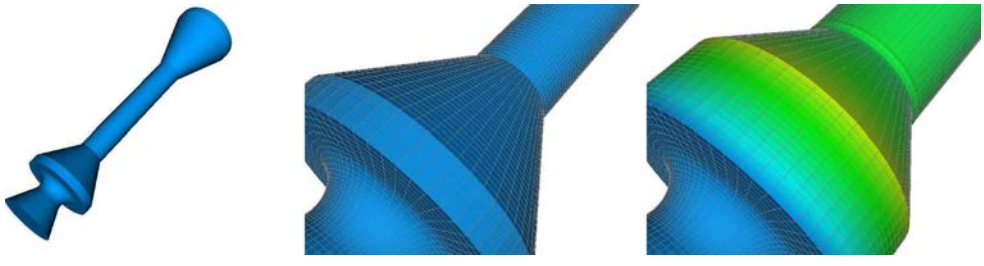


Fig. 11 Simulation of burning of Attitude Control Motor along the case with block-structured meshes using face-offsetting. *Left* subfigure shows initial geometry; *middle and right* subfigures show meshes of initial geometry and after 30% burn, respectively. *Colors* indicate magnitude of total displacements of vertices

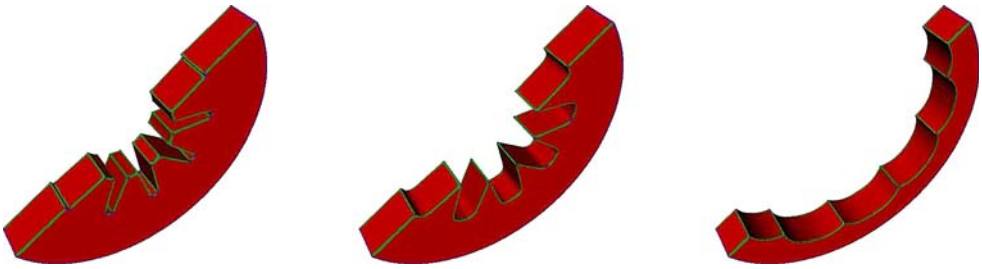


Fig. 12 Simulation of uniform burning of section of star grain of solid rocket using face-offsetting and mesh repair. *Green curves* indicate ridges in evolving geometry

physical domains, and boundary (or jump) conditions must be exchanged periodically among them to conduct a coherent simulation. In this context, the orchestration of the interactions among different modules poses a series of challenges. First, for modularity and extensibility, a physics module should be as independent as possible, so that it is transparent whether the module is running in a coupled or standalone mode, and what physics solver is being used at the other end of a coupled simulation. Second, the enforcement of jump conditions, such as conservation of mass, momentum, and energy, may require sophisticated manipulation of buffer data and involve complex buffer management. Third, the numerical coupling algorithms may be very difficult to analyze theoretically, and therefore the orchestration module must be flexible and systematic enough to support rapid prototyping of different schemes, and provide aids for developers to debug and gain insights of different schemes.

To meet these challenges, we have developed *Rocman* a control and orchestration module to coordinate multiple physics modules in coupled simulations and provide facilities to extend and implement new coupling schemes. *Rocman* is the front-end of the coupled code that directly interacts with end-developers of coupled simulations. It encapsulates the manipulation of boundary data involved in the jump conditions and the interactions between the applications. This is not only a good software design, but also enables isolating the applications to the extent that one physics module can be removed from a simulation (in the sense of not being active) without influencing the other(s), which in turn allows step-wise integration and eases debugging. *Rocman* is a high-level infrastructure, built on top of the *Roccom* integration framework. With a novel design using the idea of action-centric specification and automatic scheduling of reusable actions to describe the intermodule interactions, *Rocman* facilitates the diverse needs of different applications and coupling schemes in an easy-to-use fashion.

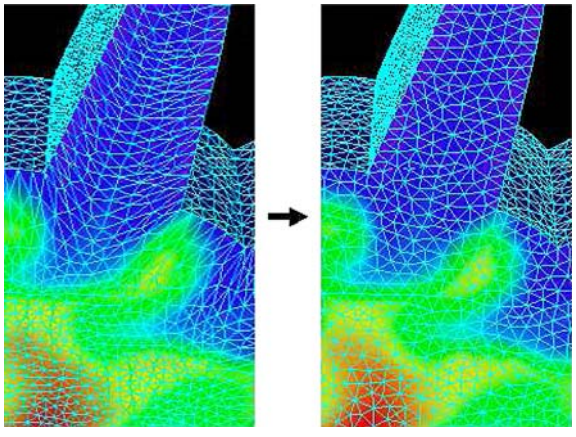


Fig. 13 Example of remeshing and data transfer of deformed star grain

5.1 *Rocman* components

Rocman contains five types of key components: top-level iterations, agents for physics modules, actions, schedulers, and coupling schemes.

One of the major tasks of *Rocman* is to drive the simulation. For this purpose, it provides *top-level iterations* including *time-marching schemes* for both steady and unsteady simulations. In the driver code, *Rocman* invokes time integration of the coupling scheme by passing in the current time and obtaining a new time, until the system reaches a designated time or a converged state.

An *agent* serves a physics module. It represents a domain-specific simulation (fluid, solid, or combustion) in a coupling scheme. The most basic task of an agent is to initialize the physics module and manage its persistent buffer data for use during intermodule interactions on behalf of the physics module using the windows and partial-inheritance data abstractions of *Roccom*.

Interactions between physics modules are encapsulated in *actions*. An action is a functional object implementing a designated calculation. An action also defines the input data, on which it operates and the

output data produced by the calculation. It typically invokes a sequence of calls to service utilities via *Roccom*'s function invocation mechanism.

A *scheduler* is a container of actions, and is responsible for determining the orders of initialization, execution, and finalization of its actions. A scheduler provides a procedure add_action() to its user for registering actions. After all the actions have been registered with a scheduler, the scheduler can then automatically schedule these actions based on the data flow among actions. The automatic scheduling constructs a call graph, which is a directed acyclic graph (DAG) for the actions, in which each edge between a pair of actions is identified by the data passing from one action to the other. This automatic scheduling of actions greatly simplifies the work of an application developer, who now needs to be concerned about only the data movement among actions without having to worry about the order of its execution. Furthermore, constructing a call graph of actions exposes parallelism among actions and potentially enables concurrent execution of all independent actions that have their input data ready. In the future, we plan to extend the run-time scheduling to allow concurrent execution of actions.

A *coupling scheme* is composed of a number of agents and a scheduler. The scheduler determines the orders that must be followed for invoking initialization, execution, and finalization of agents and actions. The coupling scheme is the only code an end-developer of a new coupling scheme must write. *Rocman* provides a rich set of predefined basic actions, which can then be used as building blocks for new coupling schemes.

5.2 Coupling scheme visualization

Understanding and debugging a complex coupling scheme poses a great challenge for a user when a variety of *schedulers* and *actions* are involved. *Rocman* provides a visualization tool that displays the data flow of actions to help users comprehend and debug coupling schemes. When a coupling scheme is constructed, an output file is generated that describes the coupling scheme and its schedulers and actions in the Graph Description Language (GDL). The output file can then be visualized by tools such as AiSee (http://www.aisee.com).

As a concrete example, Fig. 14 illustrates a simplified fluid and solid coupling scheme with subcycling of individual physics modules. In a ''system time step'', the tractions are first transferred from the fluids interface mesh onto the solids interface mesh (step 1), and a finite-element analysis of elasticity is then

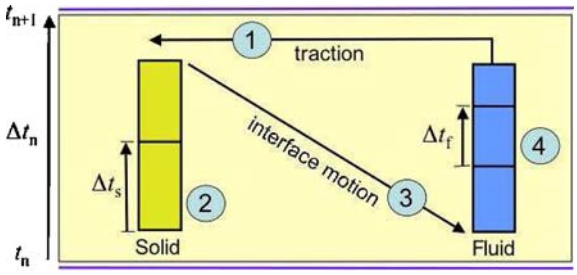


Fig. 14 Illustration of simplified time stepping scheme for solid–fluid interaction

performed to compute the displacements of the interface (step 2). During the process, the solids module may perform multiple smaller time steps based on its stability limit, and obtain jump conditions (tractions) from *Rocman*, which performs interpolation in time. After the solids module reaches the designated system time step, *Rocman* transfers the displacements of the interface (step 3). The fluids module then solves for tractions by obtaining mesh motion and solids velocity as boundary conditions (step 4).

Figure 15 shows the visualization of this simplified coupling scheme. In the graph, each node represents an *action* or a *scheduler* (a container of actions), corresponding to the steps in the above description of the coupling scheme. Each edge represents the execution order of actions and is labeled with data passed between actions. This figure was generated automatically using the GDL output of *Rocman*, except for the circled numbers, which were added manually. A scheduler node can be unfolded in AiSee graph viewer to reveal the details of the actions that the scheduler contains. This visualization capability helps development of new coupling schemes by allowing them to be debugged visually at a high level.

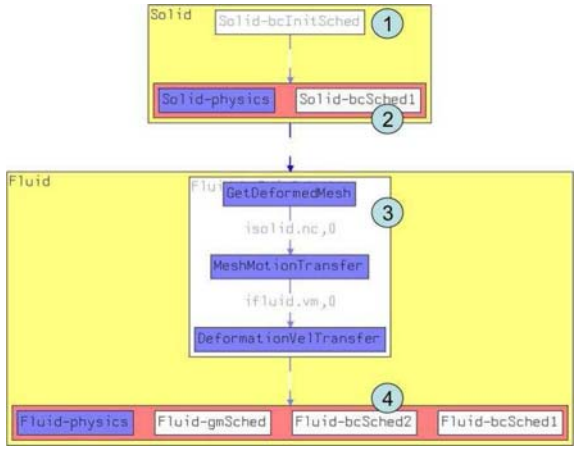


Fig. 15 Sample visualization of fluid–solid coupling scheme using aiSee

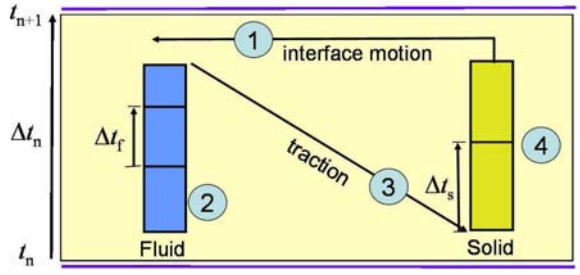


Fig. 16 Illustration of simplified time stepping scheme for fluid–solid interaction

This orchestration framework makes it very easy to experiment with new coupling algorithms, while retaining the clarity of the overall control flow. As an example, a slightly modified coupling scheme that performs fluid dynamics first can be defined as in Fig. 16. The corresponding visualization of the scheme is shown in Fig. 17. In this implementation of the new coupling scheme, only the execution order of the actions is changed so that fluid is solved before solid, while all actions are reused. This greatly simplifies quick prototyping of new coupling schemes.

6 Performance results

An indirect function invocation through *Roccom* is about two orders of magnitude more expensive than direct invocation of a function call (about 7.5 μs vs 15 ns on an IBM POWER3 SP), which is comparable with other frameworks, such as CCA [2]. The overhead of accessing the metadata of attributes through *Roccom* is also of about the same order. Because the

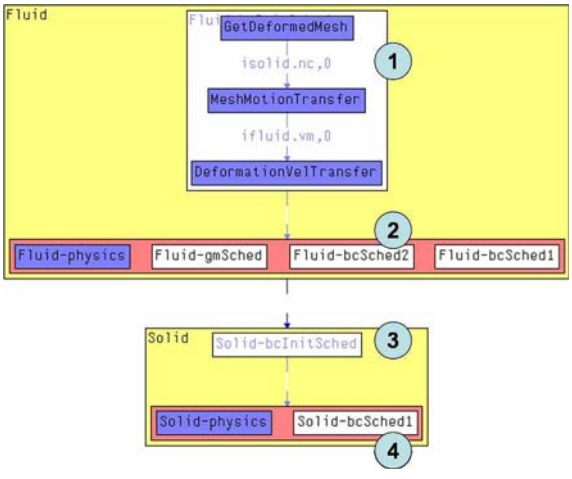


Fig. 17 The visualization of the fluid–solid coupling scheme using aiSee

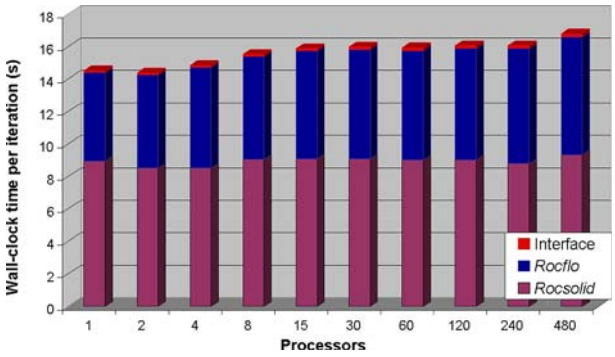


Fig. 18 Scalability of *Rocstar* with *Rocflo* and *Rocsolid* on IBM SP

granularity of computations in multiphysics simulations is usually relatively large (typically on the order of tens of milliseconds or higher), the overhead of data management and calling mechanism is negligible. In a parallel environment, *Roccom* itself does not incur spurious interprocess communication, and hence an integrated system should deliver good efficiency if the individual components are efficient.

To demonstrate the above claim of efficiency, we measure the scalability of *Rocstar* using a *scaled* problem, i.e., the problem size is proportional to the number of processors, so that the amount of work per process remain constant. Ideally, the wall-clock time should remain constant if scalability is perfect. Figure 18 shows the wall-clock times per iteration using explicit–implicit coupling between *Rocflo* (a structured fluid code) and *Rocsolid* (an implicit solid code) with a five to one ratio (i.e., five explicit fluid time steps for each implicit solid time step), up to 480 processors on ASC White (Frost), based upon IBM's POWER3 SP technology. Figure 19 shows the wall-clock time for explicit–explicit coupling between *Rocflu* (an unstructured fluid code) and *Rocfrac* (an explicit solid code), up to 480 processors on ALC. In

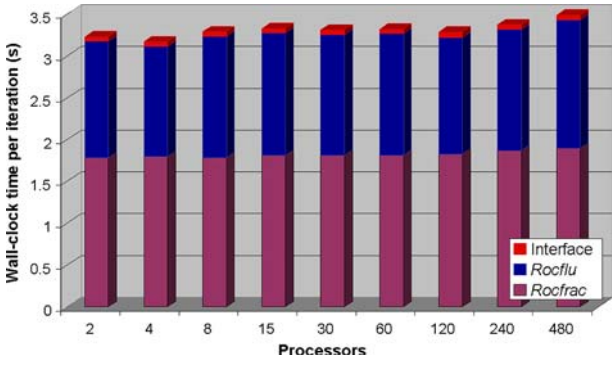


Fig. 19 Scalability of *Rocstar* with *Rocflu* and *Rocfrac* on Linux cluster

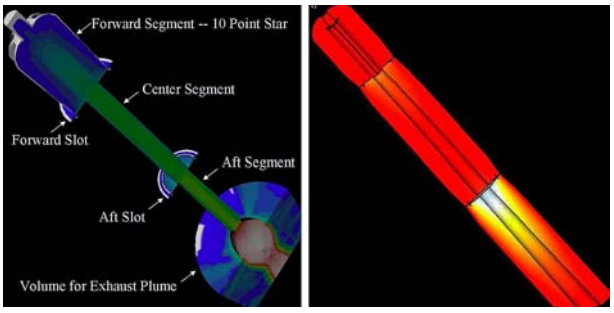


Fig. 20 Titan IV propellant slumping. *Left* cutaway view of fluids domain. *Right* propellant deformation after 1 s

both cases, the scalability is excellent even for very large numbers of processors. The interface code, dominantly data transfer between fluid and solid interfaces, takes less than 2% of overall time. Times for other modules are negligible and hence are not shown.

To demonstrate the benefits of virtualization using AMPI, we compared the performance of *Rocstar* using AMPI and MPICH/GM on different numbers of processors of the Turing Apple cluster with Myrinet interconnect at CSAR. Our test used a 480-processor dataset of the Titan IV SRMU Prequalification Motor #1. This motor exploded during a static test firing on 1 April 1991 due to excessive deformation of the aft propellant segment just below the aft joint slot [30]. Figure 20 shows a cutaway view of the fluids domain and the propellant deformation, obtained from *Rocstar*'s 3-D simulations at nearly one second after ignition for an incompressible neoHookean material model. We ran *Rocstar* using AMPI (implemented on the native GM library) on various number of physical processors ranging from 16 to 480 processors, and ran the same simulation with MPICH/GM on 480 processors. Table 1 shows the wall-clock times per iteration. The AMPI-based run outperformed the MPICH/GM-based by about 12% on 480 processors, demonstrating the efficiency of our AMPI implementation directly on top of the native GM library. Note that even better performance was obtained on 240 processors with two AMPI threads per physical processor. This virtualization allowed the AMPI runtime system to dynamically overlap communication with computation to exploit the otherwise idle CPU cycles while reducing inter-processor-communication overhead for the reduction in the number of physical processors, leading to a net performance gain for this test.

Table 1 *Rocstar* performance comparison of 480-process or dataset for Titan IV SRMU rocket motor on Apple cluster

	AMPI						MPI
Processors	16	30	60	120	240	480	480
Time(s)	15.33	8.41	5.02	3.01	1.66	2.415	2.732

7 Conclusion

In this paper, we presented a hierarchical software framework for integration of coupled multiphysics simulations. The framework is composed of an object-oriented integration interface, a set of computational and middleware service utilities, and a high-level domain-specific orchestration module. This framework facilitates integration of independently developed software modules, allows different software components to evolve relatively independently of each other, and enables rapid prototyping of various coupling schemes. The data abstractions of the framework also simplify adapting object-oriented software modules to use AMPI and take advantage of processor virtualization transparently for better parallel performance. This software framework demonstrated great efficiency in the *Rocstar* suite for detailed whole-system simulation of solid rocket motors, while greatly improved programming productivity.

Acknowledgments We thank many of our colleagues at CSAR, especially Damrong Guoy, Xiaosong Ma (now at NCSU), and Soumyadeb Mitra for their contributions to *Roccom* and service utilities, and Prof. Philippe Geubelle, Drs. Robert Fiedler, Luca Massa, Ali Namazifard, and Bono Wasistho for their input on the *Rocman* orchestration framework. The CSAR research program is supported by the U.S. Department of Energy through the University of California under subcontract B523819.

References

1. Allen G, Dramlitsch T, Foster I, Karonis N, Ripeanu M, Seidel E, Toonen B (2001) Supporting efficient execution in heterogeneous distributed computing environments with Cactus and Globus. In: Proceedings of Supercomputing '01 (CDROM), Denver, CO
2. Allan B, Armstrong R, Wolfe A, Ray J, Bernholdt D (2002) The CCA core specification in a distributed memory spmd framework. Concurr Comput Pract Exp 5:323–345
3. Budge KG, Peery JS (1998), Experiences developing ALEGRA: a C++ coupled physics framework. In: Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing
4. Bassetti F, Brown D, Davis K, Henshaw W, Quinlan D (1998) Overture: an object-oriented framework for high performance scientific computing. In: Proceedings of Supercomputing '98 (CDROM), San Jose, CA
5. Reynders JVW et al (1996) POOMA: a framework for scientific simulations on parallel architectures. In: Wilson GV, Lu P (eds) Parallel programming using C++, MIT, pp 553–594
6. Stewart JR, Edwards HC (2004) A framework approach for developing parallel adaptive multiphysics applications. Finite Elem Anal Des 40:1599–1617

7. Liskov B, Castro M, Shrira L, Adya A (1999) Providing persistent objects in distributed systems. Lect Notes Comput Sci 1628:230–257
8. Hakonen H, Leppanen V, Raita T, Salakoski T, Teuhola J (1999) Improving object integrity and preventing side effects via deeply immutable references. In: Proceedings of Sixth Fenno-Ugric Symposium on Software Technology, FUSST'99, pp 139–150
9. Stevens P, Pooley R (1999) Using UML: software engineering with objects and components. Addison-Wesley
10. London K, Dongarra J, Moore S, Mucci P, Seymour K, Spencer T (2001) End-user tools for application performance analysis using hardware counters. In: International Conference on Parallel and Distributed Computing Systems
11. Huang C, Lawlor O, Kalé LV (2003) Adaptive MPI. In: Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03), College Station, TX
12. Huang C, Zheng G, Kumar S, Kalé LV (2006) Performance evaluation of adaptive MPI. In: Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006
13. Kalé LV (2002) The virtualization model of parallel programming: runtime optimizations and the state of art. In: Los Alamos Computer Science Institute (LACSI) 2002, Albuquerque, NM
14. Zheng G (2005) Achieving high performance on extremely large parallel machines. PhD Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign
15. Kale LV, Zheng G, Lee CW, Kumar S (2006) Scaling applications to massively parallel machines using projections performance analysis tool. In: Future generation computer systems special issue on: large-scale system performance modeling and analysis, vol. 22:347–358
16. Kalé LV (2004) Performance and productivity in parallel programming via processor virtualization. In: Proceedings of the 1st International Workshop on Productivity and Performance in High-End Computing (at HPCA 10), Madrid, Spain
17. Zheng G, Lawlor OS, Kalé LV (2006) Multiple flows of control in migratable parallel programs. In: Proceedings of the 8th Workshop on High Performance Scientific and Engineering Computing (HPSEC-06), Columbus, Ohio
18. Jiao X, Heath MT (2004) Overlaying surface meshes, part I: algorithms. Int J Comput Geom Appl 14:379–402
19. Jiao X, Heath MT (2004) Overlaying surface meshes, part II: topology preservation and feature detection. Int J Comput Geom Appl 14:403–419
20. Farhat C, Lesoinne M, LeTallec P (2004) Load and motion transfer algorithms for fluid/structure interaction problems with non-matching discrete interfaces: momentum and energy conservation, optimal discretization and application to aeroelasticity. Comput Meth Appl Mech Eng 157:95–114
21. Bernardi C, Maday Y, Patera AT (1994) A new nonconforming approach to domain decomposition: the mortar element method, In: Brezis H, Lions JL (eds) Nonlinear PDEs and Their Applications, Collège de France Seminar, vol XI, pp 13–51
22. Belgacem FB, Maday Y (1997) The mortar element method for three dimensional finite elements. RAIRO Math Model Numer Anal 31:289–302
23. Jiao X, Heath MT (2004) Common-refinement based data transfer between nonmatching meshes in multiphysics simulations. Int J Numer Meth Eng 61:2401–2427
24. Jaiman RK, Jiao X, Geubelle PH, Loth E (2005) Assessment of conservative load transfer for fluid-solid interface with non-matching meshes. Int J Numer Meth Eng 64:2014–2038
25. Osher S, Fedkiw R (2003) Level set methods and dynamic implicit surfaces. Springer, Berlin Heidelberg New York
26. Sethian JA (1999) Level set methods and fast marching methods. Cambridge University Press, Cambridge
27. Thompson JF, Soni BK, Weatherill NP (eds) (1999) Handbook of grid generation. CRC Press, Boca Raton
28. Freitag L, Leurent T, Knupp P, Melander D (2002) MESQUITE design: issues in the development of a mesh quality improvement toolkit. In: 8th Int. Conf. Numer. Grid Gener. Comput. Field Sim., pp 159–168
29. Lawlor OS, Kalé LV (2002) A voxel-based parallel collision detection algorithm. In: Proceedings of International Conference Supercomputing, pp 285–293
30. Wilson WG, Anderson JM, Vander Meyden M (1992) Titan IV SRMU PQM-1 overview. AIAA Paper 92-3819