A NEW APPROACH TO SOFTWARE INTEGRATION FRAMEWORKS FOR MULTI-PHYSICS SIMULATION CODES*

Eric de Sturler, Jay Hoeflinger, Laxmikant Kale, Milind Bhandarkar
University of Illinois at Urbana-Champaign
Urbana, IL, USA

Abstract Existing software integration frameworks typically require large manual rewrites of existing codes, or specific tailoring of codes written to be used in the framework. The result is usually a special-purpose code which is not usable outside of the framework. In this paper, we propose an alternative to that model - a framework that requires little hand-modification of the programs which use it. Our proposed framework is compiler-based, mesh-aware, numerics-aware, and physics-aware. Descriptions of the codes and the system make it possible to semi-automatically generate interfacing code, while the internal parallelization, communication and numerical solution methods are left intact. We report on some preliminary experiments with an automatic load balancing framework that demonstrate the feasibility of this approach.

Keywords: component architectures, software integration, automatic load balancing

1. INTRODUCTION

It has become increasingly clear in the large-scale simulation community that developing efficient programs for complex simulations on large parallel computers is a laborious and difficult task. Therefore, several frameworks have been developed to ease the implementation of large-scale, parallel, simulations, such as POOMA [3], Overture [7], SAMRAI [12], ALEGRA [6], ALICE [1], and SIERRA [22]. These frameworks generally blend innovations in computational techniques with innovations in software technology; however, typically focusing on a few

*This work was supported in part by the US Department of Energy through the University of California under Subcontract number B341494.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35407-1_22
R. F. Boisvert et al. (eds.), *The Architecture of Scientific Software*
© IFIP International Federation for Information Processing 2001

techniques and applications. Since these frameworks simplify the implementation of parallel applications, it was assumed that these frameworks would be the right platforms for implementing the combined simulations of physical processes. However, although the efforts of constructing these frameworks in themselves have been successful, none of these frameworks has attracted a large user base or been widely adopted outside their field of application. A major concern of the application community is the required complete conversion of their software to make use of one of these frameworks, which means rewriting, loss of control over many aspects of their software, and the code's resulting dependence on the existence of and continued support for the framework. Moreover, no framework supports the wide variety of discretization schemes and numerical techniques that exist, and combining codes from different frameworks is still hard.

The Common Component Architecture (CCA) [2] and initiatives like PAWS [4] have been started to address this problem. Most research focuses on either the general aspects of components (CCA) or the development of very specific components, such as linear solvers (ESI) [8] and their interface to finite element programs (FEI) [10].

We aim to apply component principles at the level of whole applications, so that parallel applications can run both stand-alone and with other applications in our programming environment. The changes to existing parallel programs should remain minimal.

In this paper, we will describe our first experiments with this approach using the Charm++ [14] parallel runtime system in the Center for Simulation of Advanced Rockets.

We are building an integrated multi-physics Rocket Code from several stand-alone pieces, including a computational fluid dynamics code (ROCFLO), and a structural analysis code (ROCSOLID). These codes are tied together using an interface code (ROCFACE). Jointly these codes are used to simulate solid propellant rockets. ROCFLO solves the equations describing the core flow in the inner part of the rocket. ROCFLO also models the combustion of the propellant, at present (while a full 3D combustion code is being developed). ROCSOLID solves the equations describing the movement (deformation) of the solid propellant, liner, and casing. ROCFACE takes care of the transfer of data and the necessary (conservative) interpolation of physical values (temperature, pressure, and displacement) between these two applications. Since the two applications were originally constructed independently for different purposes, they contain disparate meshes. The transfer of data between these meshes involves finding a matching between adjacent elements on

the two meshes, and determining the functions that need to be used to interpolate values between those elements.

In the future, we hope to be able to integrate additional codes to handle specific situations within the rocket. A 3D combustion code may be added to the system. A *crack propagation* code will be used to model cracks that sometimes open up within the solid propellant. These are important because if a pressurized crack reaches the rocket casing, the combustion can burn through the case, causing the rocket to explode. A code simulating the paths of particles of aluminum, released from the solid propellant, which enter and burn within the core flow of the rocket, is being prototyped. Other capabilities may be implemented as stand-alone codes or additions to existing code, such as models of the mechanical joints in the rocket case, turbulence within the core flow, and the ablation of the rocket nozzle.

Although our approach seems to be an effective way to solve highly complex problems, separate the concerns of simulations of different physical processes, and preserve past effort in developing simulation programs, we know of no group that has pursued it as a systematic approach. That is, to our knowledge, no programming environment has been developed, to date, that couples, with minimal changes, existing stand-alone applications.

Several groups have worked on the mathematical and physical aspects of simulating two interacting processes (such as fluid-structure interaction) by simulating each part separately and solving the combined problem by forcing consistent solutions on the interfaces. This is referred to as a partitioned solution procedure [9]. However, this has only been done for specific and special problems, and the main results are specific, detailed schemes for those problems. Probably these schemes can be extended to other problems, and this has been pointed out [19]. However, again it seems a general programming environment that supports combining multiple grid-based applications for complex multi-physics simulations, especially involving highly dynamic adaptive simulations, has not been studied or built.

1.1. THE INTEGRATION OF NUMERICAL METHODS

The efforts in the CCA and other component projects aim at exchanging data without any semantics (meaning) associated with the data. These projects address flexible, but *raw*, data transfer. The aim of these standards is to be able to carry out the exchange of data between as wide a class of components as possible by setting standards for generic mech-

anisms and by encouraging components to support multiple formats for data exchange.

Although this is important and should greatly enhance re-usability and inter-operability, it is not enough to really ease the composition of multiple numerical codes that were not developed together or with a different use in mind. Moreover, in order to reach true plug-and-play for numerical components, as advocated in [2], especially without the need to bring all specialists together for each extension of the simulation code, we need an environment that is aware of the interaction between numerical methods.

On the other hand, existing component projects which do deal with the semantics of data at the interface, such as the FEI (Finite Element Interface), are very limited in scope (and they do not share a strict definition of components with the CCA group). They address only one important coupling: finite element codes with linear solvers.

2. A NEW TYPE OF INFRASTRUCTURE FOR COUPLED SIMULATIONS

The emphasis in this paper is on an infrastructure to facilitate the implementation of multi-physics simulations. Addressing the special requirements of coupled applications significantly complicates the implementation, and we briefly outline the most important functionality here. We divide the functionality into three major categories:

- Application coupling technology,
- Orchestration of multiple independent simulation programs with highly dynamic behavior,
- Computational Steering (physical model, mathematical model, and performance).

Without going into great detail, we would like to outline here some of the most important issues for partitioned, coupled simulations.

2.1. APPLICATION COUPLING TECHNOLOGY

The coupling of applications in a partitioned simulation involves four major issues: mesh matching, physically and mathematically consistent mapping of boundary data, the coupling of the separate solution processes in each application, and the coordination of the separate time-stepping procedures.

Generally, the constituent applications come with their own meshes, discretizations, and internal data structures. As the meshes may differ in type from application to application, they may not be aligned or even coincide at the *physical* interface. Moreover, these meshes will typically have been partitioned independently for parallelization. In addition, they may differ significantly in spatial resolution. As a result, the transfer of data in a physically and mathematically correct way is very complicated.

We need to compute a matching between the meshes of interacting applications. This matching will indicate the interaction between interdependent applications at the level of individual elements or cells. Together with the equations that must be satisfied on the boundary, constraints on the mapped values, and possible conditions to be satisfied, the results of the mesh matching will define the mapping of variables between applications [18, 13, 15, 19].

After identifying which parts of separate meshes correspond, we need to define mappings of those variables that are needed to make the mathematical model in the *neighbor* application well-defined. The physics of the underlying application or mathematical stability constraints may require that certain functions of mapped variable fields be conserved. This can be relatively simple as in the conservation of a single variable like mass, or it can be more complicated as in the conservation of an integral over a function of multiple variables, such as work expressed as the product of displacement times force, and momentum expressed as mass times velocity [5, 9]. However, even the conservation of a single variable may not be trivial if we map variables between meshes that differ in type of discretization (finite element versus finite volume), element type (tetrahedral vs hexahedral), finite element basis functions, or if meshes vary widely in spatial resolution (note that there may be accuracy constraints). In addition, some variables may have specific constraints. For example, a mass value would be required to be non-negative.

In the partitioned approach we run each application separately, and they interact through their boundaries. If we require at each (major) time step that values on a shared boundary be consistent among applications, then we call such an approach strongly coupled [9]. In this case we have to decide on the tolerances on the convergence across applications. If we do not require this consistency we call the approach loosely coupled [20]. We envision using both approaches. These choices have an influence on the overall solution accuracy and efficiency. A potential problem that arises in the strongly coupled procedure is divergence or stagnation of convergence across multiple applications. Even though in

each step all applications converge, their shared representation of the (fields of) variables on one or more boundaries may not.

The issues involved in setting time steps are closely linked with orchestration, but there are also independent issues. We need a sequence of global (maximum) time steps at which each application must deliver a solution. In the case of a strongly coupled approach, we require consistency at these global time steps. Within the limits of a global time step, each application can choose its own time steps. The global time steps will generally be determined by accuracy constraints, the time-scale of relevant physical phenomena, or a global CFL-like condition.

2.2. ORCHESTRATION

The orchestration of a collection of applications defines at a high level how the various simulations interact, how each does its time integration, and how these different time integration schemes are combined. Furthermore, several problems can arise while running such a multi-application simulation in a partitioned approach. Convergence problems may occur, in a single application or across multiple applications. Moreover, the environment may have to dynamically start additional applications, such as turbulence in a Computational Fluid Dynamics (CFD) application, or crack propagation in a structural application, and dynamically swap applications if required by the simulation. The programming environment must orchestrate highly dynamic interactions of parallel partitions of applications, based on assumptions of applications, requirements on boundary exchanges, and convergence across applications.

2.3. STEERING

Since simulations may run for long periods of time on parallel supercomputers, we must be able to interact intelligently with these simulations and maybe spin-off additional simulations derived from intermediate results of the main simulation. It may also be necessary to adjust load balancing and parallelization schemes periodically, to optimize the performance while the application is running or to adjust these schemes to changes in the computational environment. Therefore, our programming environment needs to include steering, both from a mathematical/physical point of view and from a computational/performance point of view [16, 21]. Given that these programs will run for very long times, it is unlikely that the results will be continuously monitored. Hence, we envision the use of smart, event-driven check pointing. Based on a description of the relevant states of the simulated processes, or of the relevant dynamic behavior, the programming environment should be able

to save the necessary data when triggered by the appropriate condition. The check-pointed data could be used later to run the simulation with higher precision or with more appropriate models.

Conceptually, a control channel would be kept open between the codes, facilitating orchestration and steering. This channel would allow a code to request services from, and report conditions to, the framework and other codes. This control channel would allow the framework to "call" the individual codes at the appropriate times, initiate data transfers, start codes in response to dynamic events, and handle exceptional situations.

3. AUTOMATIC TRANSLATION INTO A FRAMEWORK

The simulation codes should be independent, each capable of running by itself, but also able to cooperate with others when used within the framework. To avoid re-writing a code to fit into the framework, we propose the use of automatic tools which could do the translation required to allow the code to fit into the framework.

The translation of a given code would be guided by a Code Description (CD). The CD would essentially be an interface specification for the code, giving all the information necessary to

- locate data within the code which would be available for other codes,
- drive the conversion of data from one code to another,
- define the conditions placed on data passed to the code from the outside, and
- describe special functionality of the code which might be used under exceptional conditions.

An Orchestration Description (OD) would be used to describe how the various component codes should interact. It would describe which modules should exchange data, how to convert the data from one component to the other (including the matching of mesh points [13]), conditions under which certain modules should be invoked, a system-wide coordinate system for the various meshes, and any system-wide constraints or information which spans components. Trigger conditions could be spelled out in this specification which show how unusual situations should be handled, when it is necessary to switch to a different simulation code, and when check-pointing should occur.

The CD and the OD could take the form of a set of annotations, interspersed with the statements of the simulation code itself, such as the structured comments of OpenMP [17]. It could also take the form of a separate description file, written in some language, such as Python [23]. The structured comment approach has the advantage that a direct correspondence could be drawn between the description and the code itself. The separate description file has the advantage that it pulls all the pieces of the description together in one place.

The CD would guide the placement of data ports [2] within a given code, allowing the code to communicate the values of certain variables with other codes and the Framework. The Framework would employ interpolation functions appropriate to the quantities and numerical methods described in each code's CD.

The OD would guide the automatic creation of a "driver" routine for the simulation, calling the individual components in the proper order and with the proper parameters. The driver routine would contain the proper convergence criteria, and tests for the trigger conditions, as specified in the OD.

A well-known guiding principle from Software Engineering is to automate those programming tasks that can be best done by the computer, while giving human programmers the tools to carry out tasks best done by them. The CDs and the OD for a simulation system would give the programmer a human-friendly way of describing the complex codes to be combined, and how to combine them.

The framework would employ compiler technology to analyze each component code, the CDs and the OD. It would then carry out the tedious transformation to automatically generate the combined code.

4. PRINCIPLES OF APPLICATION INTEGRATION

To guide the implementation of our model of application integration, we state a set of general principles. A framework which adheres to these principles should be able to couple multiple stand-alone application codes successfully.

Cooperative Interoperability Principle: "*Different stand-alone application modules should coexist as a part of a single simulation.*" The codes should execute concurrently and exchange the results of their computations without human intervention. The framework would not impose its own restrictions on the parallelization or memory usage of the codes. It should accept whatever code optimizations are used by each code.

Minimal Source Change Principle: "*The changes required of a stand-alone simulation code for use within a framework should be minimal.*" This principle addresses the software engineering issues of using an integration framework. The idea is that there should be only a single version of a code. That code is then converted automatically for use in the framework. Development may then continue on the single version, which is converted for use in the framework whenever necessary. This lessens the error-prone work needed for converting the code by hand, allows the programmer to maintain a single version of a code. and allows the code author to retain ownership of the code.

Correct Data Exchange Principle: "*The framework should exchange data between codes in a mesh-, numerics-, and physics-aware fashion.* " This principle addresses the ease of converting data between codes. The framework should contain facilities for mesh-cell-matching and data conversion, relieving the programmer from the effort of programming these things. The framework should have a set of physically-correct, numerically stable conversion techniques already deployed.

Dynamic Adaptation Principle "*The framework should dynamically respond to exceptional situations within the simulation.*" This principle addresses the ability of the simulation system to adjust the operation of the overall simulation in a very fine-grained way, in response to dynamic situations. If the conditions within the simulation go beyond the bounds of a given code in one part of the mesh, a different code may have to be started to take over the simulation from that code, for that part of the mesh.

5. PROOF OF CONCEPT WITH A LOAD BALANCING FRAMEWORK

The class of frameworks described thus far in this paper is broad. Many frameworks could be built, based on the principles outlined in Section 4. In our own project, we have proceeded in measured steps toward the general goal of an integration framework for simulation codes. We are building on the experience of integrating our rocket simulation code by hand. The Rocket Code developers (Prasad Alavilli, Dennis Parsons, Ali Namazifard, and Jim Jiao) manually integrated the two simulation components (ROCFLO and ROCSOLID) with an interface code (ROCFACE).

However, the task of building a general multi-simulation code integration framework from scratch is a daunting one, so we have decided to do a proof-of-concept with a less ambitious task - implementing a framework for doing load-balancing for our hand-integrated Rocket Code. The

basis of the load-balancing system is the Charm++ system, developed by Kale, et al.

Toward that end, we first chose to implement the hand-integrated rocket simulation code on top of the Charm++ system. We chose to use Charm++ as a substrate for our integration framework because of its support for automatic interleaving of multiple components, and because of its dynamic load balancing abilities. We developed a methodology for converting this code to the Charm++ system that we believe is automatable, so that a compiler-based tool could be built for automatically converting a code for use with Charm++. This automation of the conversion process will satisfy the Minimal Source Change Principle.

In the following sections, we will briefly describe the Charm++ system, and how we chose to use it with our Rocket Code.

5.1. THE CHARM++ SYSTEM

Charm++ is an explicitly parallel object-oriented system. Charm++ programs are typically written using C++, and use a small interface description language, along with the Charm++ runtime support system (RTS).

The basic entity in Charm++ programs is a data-driven object. A computation comprises many such objects (or indexed collections of such objects), which are mapped to processors under the control of the Charm++ RTS. Such objects have a global, system-wide ID, and can communicate with each other via asynchronous method invocations using these IDs. As the IDs remain the same, even when the RTS migrates objects from processor to processor, the application-writer can write their code without concerning themselves with load balancing (i.e. they write the code for one object to communicate with the other without concerning themselves with where these objects live).

The core of the Charm++ RTS consists of a message-driven scheduler. Messages in Charm++ represent computations to be performed (methods to be invoked on data-driven objects). The scheduler repeatedly chooses messages from a processor-wide pool and executes methods denoted by them. Thus, messages directed at different objects (possibly belonging to different modules) are interleaved allowing concurrency across different components on the same processor.

The data-driven objects provide a natural "grain" of execution to be monitored for possible load imbalance. The Charm++ RTS incorporates a load balancing support module, which keeps track of execution times for each object, and communication patterns among objects. These statistics are then provided to a "plug-in" load balancing strategy

module that decides whether and how to remap these objects among processors, to get better load balance.

5.2. OUR APPROACH TO USING CHARM++

The component codes, ROCFLO and ROCSOLID, were both written in Fortran 90, using MPI to implement parallelism and message passing. MPI forces the user to identify processors with integers representing the processor numbers. With MPI, the number of MPI processes is equal to the number of processors. To connect the MPI code with Charm++, we chose to replace the MPI runtime library with a library implemented on top of Charm++. In this form, the integers in the source code no longer represent processor numbers, but instead indicate *chunk* numbers.

A *chunk* in this context refers to the combination of a thread of execution and its data. In the context of an MPI program, a chunk is similar to an MPI process, but without the separation of address spaces that is normally present with MPI processes.

By doing this, we decouple the application code from a specific number of processors, and decouple a specific chunk from a specific processor. Then, Charm++ is free to allocate more chunks than processors, and move chunks around from processor to processor, if load-balancing is required.

5.3. LOAD BALANCING METHODOLOGY

The approach that we are exploring for the load balancing framework involves multi-partition decompositions. Computations in each individual module are partitioned into a large number of chunks, such that there are many more chunks than processors. The code and data for simulating each chunk is encapsulated within a data-driven object. The program is written in such a way that the objects send messages to other objects, rather than sending messages to processors. As processors are not part of the programmer's ontology, the system is free to move or migrate objects among processors, thus effecting load balancing when needed.

As multiple chunks, possibly belonging to different modules (or applications) are mapped to each processor, their execution must be interleaved by the runtime system. Data-driven interleaving, which depends on a scheduler to schedule computations of individual chunks, depending on the availability of their data (messages), accomplishes such interleaving efficiently. As the chunks are migrated from processor to processor,

their messages must be correctly forwarded. Both of these features are effectively supported by the Charm++ system.

For accomplishing load balancing, Charm++ incorporates a sophisticated load balancing subsystem. The particular strategy we use exploits the "principle of persistence": the fact that in most scientific computations, the computational loads of the chunks, and their communication patterns, are highly correlated with their values in the immediate past. This is true even for computations that require abrupt adaptive refinements, since such refinements are relatively infrequent. The load balancing framework carries out measurements of these characteristics, and then balances the load when needed, using a suite of strategies that are useful in different circumstances.

5.4. AUTOMATIC CODE CONVERSION TO THE CHARM++ FRAMEWORK

One of the challenges we faced was reconciling the need for Charm++, with our desire to make minimal changes to existing application codes. This challenge was overcome with the development of an additional run-time library. The Adaptive MPI (AMPI) library was built on top of Charm++ to provide a complete implementation of the MPI library routines. The MPI calls in the original code are intercepted by this library. With these techniques, it became possible to port the existing MPI codes, written using Fortran 90, to our run-time framework.

A few other changes to the applications were still needed within the application codes. When the MPI processes of the application codes are converted to chunks, they lose the address space separation of the original codes. This means that all globally-visible data of the original codes for the chunks executing on a single processor would be placed at the same memory locations, and interfere with each other during execution. So, such references had to be eliminated from the application codes. This is possible by dynamically allocating them at run-time, or else by statically allocating expanded versions of the global variables and indexing them by the chunk number.

In addition, subroutines that pack and unpack the chunk's private data were coded by hand. However, this process is quite mechanical, and could be completed easily once the principles were understood. ROCFLO and ROCSOLID were converted with a few days of effort, whereas ROCFACE (which was written in C++ with MPI) was converted in 45 minutes. The observations made during this conversion process, coupled with the compiler expertise in our team, led us to realize that this conversion process can be fully automated with the help of a

Table 1 Comparison of MPI and AMPI versions of ROCFLO & ROCSOLID. All times are in seconds. Note that this is scaled problem.

Processors	*ROCFLO*		*ROCSOLID*	
	MPI	*AMPI*	*MPI*	*AMPI*
1	9.0192	8.8122	18.240	17.797
8	8.0796	8.0958	18.413	18.458
16	8.1908	8.2682	18.564	18.830
32	8.3415	8.3093	19.410	18.947
64	8.5535	8.6183	19.236	19.500
128	9.4889	9.6370	19.766	20.499

compiler that can perform interprocedural analysis and source-to-source transformations. Work on such an automatic conversion program is in progress.

With these conversions, the rocket simulation programs are now ready for adaptive automatic load balancing. We expect to test these abilities in the near future, when the application incorporates such features as adaptive mesh refinements, and also when running on dynamic environments such as workstation clusters.

6. EXPERIMENTAL RESULTS

Original ROCSOLID and ROCFLO performance results were compared with their implementations on our framework. Refer to Table 1 for the performance results. Experiments were performed at National Center for Supercomputing Applications (NCSA) on an Origin2000 machine (250 MHz R10000 processors). A version of Charm++ that uses native MPI as a communication layer was deliberately chosen in order to measure the overhead that AMPI incurs over MPI. Charm++ can also be made to use shared memory arenas for communication, resulting in better performance.

In the process of conversion to our framework, the global data items used by both codes were "privatized" with respect to threads, so that multiple threads could co-exist on the same processor. Timings on one processor point to the effects of this privatization. In ROCFLO, this privatization was done by extending the dimensions of global data items, where the thread number was used as an index to access thread-private data. In ROCSOLID, we encapsulated the global data items in a single user-defined type, which is dynamically allocated by every thread at

initialization. Each data item was then accessed with an indirect reference. Surprisingly, this speeded up execution of both ROCFLO and ROCSOLID on one processor. We suspect this effect was due to coincidental rearrangement of data items, reducing cache misses. These are still preliminary results, and more thorough experiments are being performed.

It should be noted that the communication overhead due to the additional communication layer of Charm++ and AMPI is eclipsed by better cache behavior, and is less than 4% even on a higher number of processors. We suspect that the overhead on more processors is because collective operations in AMPI are not tuned for higher number of processors.

7. SUMMARY

Existing code integration frameworks have not attracted a large number of users. We believe that this is due to primarily Software Engineering issues, such as the need to rewrite a code to use a framework, the need to maintain multiple copies of a code, and the error-prone nature of recoding a working program.

We have proposed a set of code integration principles that we believe will make integration frameworks more widely accepted by the applications community, because frameworks adhering to these principles would support the plug-and-play use of a code within any of them.

A preliminary experiment has targeted codes for a Charm++ load balancing framework and we have found that automatic translation of codes for that framework is indeed feasible.

References

[1] ALICE Web page. http://www.mcs.anl.gov/alice, Mathematics and Computer Science Division, Argonne National Laboratory.

[2] R. Armstrong, D. Gannon, A. Geist, S. Kohn K. Keahey, L.C. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high performance scientific computing. In *Proceedings of High Performance Distributed Computing*, August 4-6, 1999.

[3] S. Atlas, S. Banerjee, J. C. Cummings, P. J. Hinker, M. Srikant, J. V. W. Reynders, and M. Tholburn. Pooma: A high performance distributed simulation environment for scientific applications. In *Proceedings Supercomputing '95.*

[4] Pete Beckman, Pat Fasel, Bill Humphrey, Sue Mniszewski, and MaryDell Tholburn. PAWS technical description, 1997. http://www.acl.lanl.gov/PAWS/docs/TechDescription.html

[5] C. Bernardi, Y. Maday, and A. Patera. A new nonconforming approach to domain decomposition: the mortar element method. In H. Brezis and J.L. Lions, editors, *Nonlinear Partial Differential Equations and their Applications*. Pitman, 1989.

[6] K. G. Budge and J. S. Peery. Experiences developing alegra: A c++ coupled physics framework. In M.E Henderson, C. R. Anderson, and S. L. Lyons, editors, *Object oriented methods for interoperable scientific and engineering computing, proceedings of the 1998 SIAM workshop*, October 21-23, 1998.

[7] William D. Henshaw D. L. Brown and Daniel J. Quinlan. Overture: An object-oriented framework for solving partial differential equations on overlapping grids. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, pages 58–67. SIAM, 1998.

[8] Equation Solver Interface Forum. http://z.ca.sandia.gov/esi/

[9] C. Farhat, M. Lesoinne, and P. LeTallec. 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. *Computer methods in applied mechanics and engineering*, 157:95–114, 1998.

[10] Finite Element Interface web page. http://z.ca.sandia.gov/fei/

[11] M. Hall, B. Murphy, S. Amarasinghe, S. Liao, and M. Lam. Detecting Coarse-grain Parallelism Using An Interprocedural Parallelizing Compiler. *Supercomputing 95*, December 1995.

[12] Richard Hornung and Scott Kohn. The use of object-oriented design patterns in the SAMRAI structured AMR framework. In *Proceedings of the SIAM Workshop on Object-Oriented Methods for Inter-Operable Scientific and Engineering Computing*, October 1998.

[13] X. Jiao, H. Edelsbrunner, and M. T. Heath. Mesh association: Formulation and algorithms. In *Proceedings of the 8th International Meshing Roundtable 1999, October 10-13, 1999, South Lake Tahoe, California*, pages 75–82, 1999.

[14] L. V. Kale and S. Krishnan. Charm++: Parallel programming with message-driven objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[15] N. Maman and C. Farhat. Matching fluid and structure meshes for aeroelastic computations: A parallel approach. *Computers & Structures*, 54(4):779–785, 1995.

[16] J. E. Moreira, V. K. Naik, and D. W. Fan. Design and implementation of computational steering of parallel scientific applications. In M.T Heath e.a, editor, *Proceedings of the Eight SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 14-17, 1997*, 1997.

[17] OpenMP web page. http://www.openmp.org, OpenMP Consortium.

[18] S. Plimpton, B. Hendrickson, and J. Stewart. A parallel rendez-vous algorithm for interpolatio between multiple grids. In *Proceedings of the 8th International Meshing Roundtable 1999, October 10-13, 1999, South Lake Tahoe, California*, pages 75–82, 1999.

[19] R. Löhner. Robust, vectorized search algorithms for interpolation on unstructured grids. *Journal of computational Physics*, 118:380–387, 1995.

[20] R. Löhner and C. Yang and J. R. Cebral and J. D. Baum and H. Luo and D. Pelessone and C. Charman. Fluid-structure interaction using a loose coupling algorithm and adaptive unstructured grid. In *Computational Fluid Dynamics Review 1995*, pages 755–776, New York, 1995. John Wiley.

[21] R. S. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed. Autopilot: Adaptive control of distributed applications. *Future Generation Computer Systems.* special issue (Performance Data Mining), to appear, available from http://www-pablo.cs.uiuc.edu/Publications/publications.htm.

[22] L. M. Taylor. Sierra - a software framework for developing massively parallel, adaptive, multi-physics, finite element codes, 1999. presentation at the International conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), June 28 - July 1, Las Vegas, Nevada, USA.

[23] G. van Rossum. Glue it all together with python. In Craig Thompson, editor, *Workshop on Compositional Software Architectures, Workshop Report, Monterey, California, January 6-8, 1998*, 1998.

DISCUSSION

Speaker: Milind A. Bhandarkar

Morven Gentleman : How much of your approach depends on the open context in which RCSOLID and ROCFLO were developed? Would it be possible in a context where, perhaps for proprietary reasons, the code, designs, data structures, algorithms, etc. were not so exposed?
Eric de Sturler : [*This reply was provided after the conference. – Ed.*] First of all, the prime intention of this framework is to help collaborating researchers to combine their codes for multiphysics simulations with minimal efforts and in a short time frame. To this end the code offers functionality to exchange boundary data, to match disparate meshes, coordinate timesteps, and compute physically correct solutions of the joint physical processes taking place on the boundary. The mathematics and physics needed for a correct time integration generally needs to be written for the specific applications, but can then be implemented easily within the framework.

Clearly, this requires the framework to have access to several types of data from each application and the associated geometric information: field variables defined on the boundary, time step constraints, the geometric information that defines the mesh on the boundary, type of finite elements/volumes, shape functions, and potentially other data. The way this has been done for currently integrated codes is that developers added a single (well-defined) module in which the data needed in other applications is extracted or computed and declared and in which the data needed from other applications is declared. This is the only data that needs to be visible to the framework. In order to (dynamically) start a partition of an application with its own data (chunk), to migrate it to another processor (load balancing), or to remove it, the framework must be able to interact with the application (partition). Some handle must be passed to the framework to allow it to signal a chunk that it will be moved. The chunk must then be able to export its data to the framework, after which it will be removed. After migrating to another processor the chunk will be restarted with its data.

Following encapsulation and separation of concerns principles the framework is designed such that the handles mentioned above should be the only features required to implement the functionality of the framework. The importance of such a design is that new codes can be added to the framework without undue changes in other codes and without significant interaction with developers of other codes. In general some interaction will be needed if new/different data is required for interac-

tion with the new code. For example new coupling algorithms may need to be added.

If one wants to integrate a code that is not available as source code, some wrappers will be needed. For example, many finite element packages are available as a library of object modules that can be called in the user's program. The internal data structures, algorithms and so on are typically unknown. Only the (public) interface is described. In this case we would write a small subroutine that has the required handles, initializes the data for the subroutine(s) that will compute the solution to the sub-problem of interest, call the library routine(s), extract the required data from the results and provide those in a form that is usable and visible to the framework.

One could think of strategies to use simulation codes that are available only as a single monolithic executable, but we think this would not be useful.