Infrastructure and Interfaces for Large-Scale Numerical Software *

Lori A. Freitag, William D. Gropp, Paul D. Hovland, Lois C. McInnes, and Barry F. Smith
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439-4844
{freitag,gropp,hovland,mcinnes,bsmith}@mcs.anl.gov

Abstract

The complexity of large-scale scientific simulations often necessitates the combined use of multiple software packages developed by different groups in areas such as adaptive mesh manipulations, scalable algebraic solvers, and optimization. Historically, these packages have been combined by using custom code. This practice inhibits experimentation with and comparison of multiple tools that provide similar functionality through different implementations. The ALICE project, a collaborative effort among researchers at Argonne National Laboratory, is exploring the use of component-based software engineering to provide better interoperability among numerical toolkits. We discuss some initial experiences in developing an infrastructure and interfaces for high-performance numerical computing.

Keywords: high-performance scientific computing, component software, nonlinear PDEs

1 Introduction

Researchers in computational science have benefited from the encapsulation of expertise in numerical libraries for many years. However, the complexity and scale of today's multidisciplinary scientific simulations imply that development work must be leveraged over many individual projects, because writing and maintaining a large custom application usually exceeds the resources of a single group. These issues, coupled with the multilevel memory hierarchies of distributed-memory architectures, create ever more challenging demands for high-performance numerical software tools that are flexible, extensible, and, perhaps most important for the discussion in this paper, *interoperable* with complementary tools. As we aim for scalable, efficient, and portable performance over the lifetimes of scientific applications, we are exploring techniques for managing software changes, which may range from evolutionary kernel improvements to paradigm shifts in algorithms and software organization. In this paper we describe recent efforts to develop a component-based approach to building numerical tools.

The term component has been used in many ways by the software community (see, e.g., [17]). Here we define a software component to be an encapsulated software object that provides a certain set of functionalities or services and can be used in conjunction with other components to build applications. A component consists of an API (application programming interface) and one or more implementations, and conforms to a prescribed behavior within a given computational framework.

The mainstream computing community has developed interoperability mechanisms (e.g.,

*The authors were supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

distributed object technology such as the COM family and CORBA, and component technology such as Enterprise Java Beans) to address similar levels of complexity within their applications. Our approach is to leverage this work when appropriate, recognizing that the features of large-scale scientific computation present different challenges and thus demand different solutions. One challenge is the need for efficient and scalable performance on distributed-memory architectures such as symmetric multiprocessors and workstation clusters. Also, the culture of research computing differs from that of the business world; scientists need to be able to explore their ideas without requiring legions of programmers to translate from scientific abstractions to actual code, and without becoming overwhelmed by details (e.g., security) that are not of primary interest. These issues are further discussed in [5]. A variety of researchers, including [2, 8, 15, 18], are considering similar issues within the context of high-performance scientific software.

Our goal is to leverage the strengths of different high-performance toolkits, *not* to develop a single massive library into which everyone contributes code. As such, we have a minimalist philosophy that focuses on developing infrastructure for dynamic assembly of software tools and designing performance-sensitive abstract interfaces that define the interactions among them.

The remainder of this paper motivates this approach and explains our design strategy. Section 2 introduces the needs of a simulation based on partial differential equations (PDEs). Section 3 presents our infrastructure design requirements and some interface issues for numerical software components. This discussion emphasizes support for multiple underlying component implementations (e.g., various mesh management techniques and algebraic solvers) and for automatically generating one type of component from another (e.g., producing a Jacobian component by computing derivatives of a nonlinear function). Preliminary observations and directions of future work are discussed in Section 4.

2 Motivating Example

While our work is motivated by collaborations with scientists studying a range of different areas, including astrophysics, microtomography, combustion, superconductivity, and molecular dynamics, we illustrate some of the challenges typical of this work with a particular example. We consider the modeling of Rayleigh-Taylor instabilities, which is part of an ongoing project by Argonne and University of Chicago researchers [16].

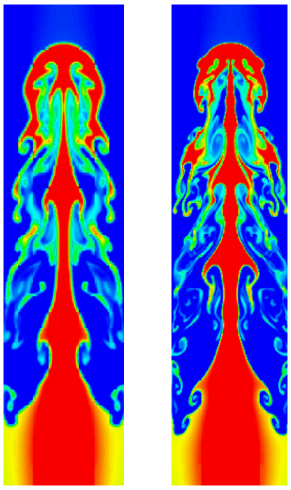

Figure 1: A single plume Rayleigh-Taylor instability modeled as a compressible fluid with the Euler equations at two different levels of refinements (left: 14,848 cells; right: 54,208 cells).

In Figure 1 we show a single plume Rayleigh-Taylor instability simulated using an adaptive Piecewise Parabolic Method (PPM) method at two different levels of refinement. The figure on the right was computed at a higher resolution than the figure on the left, and we see a corresponding increase in the fine-scale features evidenced in the numerical simulation. Because the Raleigh-Taylor phenomenon is chaotic, different discretization schemes, resolutions in the computational model, and mesh types result in different fine-scale features. Thus, an open question is “Which numerical scheme, mesh type, and resolution most accurately capture the Rayleigh-Taylor instability?”

Table 1: Methods under consideration for modeling a Rayleigh-Taylor instability.

Fluid Regime	Mathematical Model	Discretization Technique	Mesh Type	Solution Technique
Incompressible†	Navier-Stokes	Spectral	Fixed Cartesian	Explicit
Incompressible†	Navier-Stokes	Spectral Elements	Fixed Cartesian	Semi-Implicit
Compressible†	Euler	PPM	Fixed Cartesian	Explicit
Compressible†	Euler	PPM	Adaptive Cartesian	Explicit
Compressible†	Euler	Discontin. Galerkin	Adaptive Tetrahedral	Explicit
Compressible‡	Euler	Finite Volume	Fixed Cartesian	Implicit
Compressible‡	Euler	Discontin. Galerkin	Adaptive Tetrahedral	Implicit

† existing models
‡ potential future models

We are therefore studying different numerical strategies for Rayleigh-Taylor simulations (see Table 1 for a list of techniques). Because the codes in the current generation of models are all distinct from each other, we face difficulties in ensuring that they are all running the same initial and boundary conditions. An environment that allows computational scientists to interchange models, discretization techniques, and mesh management strategies from a single application code would greatly reduce the likelihood of such inconsistencies. In addition, to facilitate algorithm experimentation and comparison for implicit and semi-implicit schemes, we need support for various linear and nonlinear solvers, as well as techniques for sparse derivative computations.

3 The ALICE Project

The current generation of software infrastructure is incapable of providing this "plug and play" functionality, even if the individual tools have been built using modularity and object-oriented design. We are investigating techniques for dynamic, component-based interactions as part of a flexible architecture within the Advanced Large-scale Integrated Computational Environment (ALICE) [1]. Our holistic perspective, where we consider many facets of numerical simulations rather than simply one or two areas, is one of the key strengths of our approach. Our work builds on practical experiences with two-way interfaces between the existing parallel tools SUMAA3d [9] and PETSc [4] (discussed in [10]) and between PVODE [13] and PETSc (discussed in [5]). We also build on early experiences with common interface design for parallel linear algebra tools in the Equation Solver Interface forum [7]. The time is ripe for such activities; rich sets of unstandardized tools exist for certain functionalities (e.g., meshes and algebraic solvers), so that we can begin to exploit their differences and leverage their commonalities.

Section 3.1 discusses the development of light-weight infrastructure to manage both synchronous and asynchronous tool coordination. Section 3.2 describes recent work in enhancing existing toolkits so that they can function in a more dynamic computing paradigm.

3.1 ALICE Infrastructure

Traditional large-scale numerical simulations are almost always implemented by routines that are called in a well-defined order to implement a deterministic numerical algorithm. When object-oriented techniques are used in numerical computing, the standard approach is to encapsulate the data structures in objects, while still allowing the application programmer to call a sequence of functions that operate on the objects to perform the desired calculations.

On the other hand, the programming of graphical user interfaces and transaction processing systems has moved away from the expression of a computation as a linear list of functions that are called. Rather, (possibly distributed) objects are viewed as making requests of and serving requests of other objects.

We adopt two complementary models to handle the differences between these two types of interactions:

1. synchronous, local-address-space function calls (e.g., in C++ calling methods on objects), intended for implementing numerical algorithms; and

2. asynchronous, possibly remote, transactions such as accessing data objects for monitoring, steering, and visualization, as supported by the ALICE Memory Snooper [3]. (Here we use the term asynchronous to mean that the object may serve a request while simultaneously performing a numerical calculation.)

The primary design requirements for model 1 are that basic functionality should be as efficient as standard procedural code (Fortran/C/C++) and that no "run-time" system (e.g., threads) be needed. Our implementation, which takes the form of a "microkernel" [5] for use by ALICE-compliant tools, supports runtime binding of different toolkits, dynamic addition of methods, and object attachment through a dynamically loaded library approach. A benefit of this approach is that it promotes well-designed interfaces that are completely separate from implementations, so that various external toolkits can be introduced and the community can begin to work toward defining sets of canonical interfaces. This movement beyond layered class libraries is crucial for managing the social and technical complexities of large application projects.

Whereas model 1 meets our basic performance requirement, model 2 provides additional functionality, especially for remote operations. Model 2 requires a relatively large runtime infrastructure to support marshaling of arguments, communicating among remote processes, locking and unlocking data structures to allow access from multiple threads, and so forth. Model 2 could use technology such as Enterprise Java Beans, CORBA, or COM as implementation mechanisms.

3.2 Component Interfaces

Designing interoperable software for Raleigh-Taylor simulations and other nonlinear PDE-based models is difficult, given the modeling required to capture the physics of the problems in sufficiently realistic detail.

3.2.1 Nonlinear Function Evaluation

For example, for the explicit, semi-implicit, and implicit methods listed in Table 1, we must discretize a nonlinear function throughout the computational domain. In Figure 2 we show an example containing three base components: a mesh component, a discretization component, and a local physics component. These base components are used in combination to form a compound component that computes $F(x)$, where F is a vector containing the discretized nonlinear function, and x is the current iterate.

For each base component in Figure 2, we show example objects and interfaces that are used in computing and assembling F. For example, the mesh component contains geometric information such as the spatial locations of vertices and elements, as well as hierarchical and connectivity information. Application-specific data stored in the vector x is attached to mesh entities using an abstract interface such as `M::attachData(`x, I_x, E`)`, where M is a mesh and I_x is the mapping that relates the vector indices to the appropriate mesh entities of type E. Note that we can use introspection techniques to determine such interfaces at runtime. The local values of the vector, F_i, are computed by the local physics component, which requires information from both the mesh and the discretization components. For example, the computation of each F_i requires a discretization stencil, which we define to be the set of local mesh entities and their connectiv-

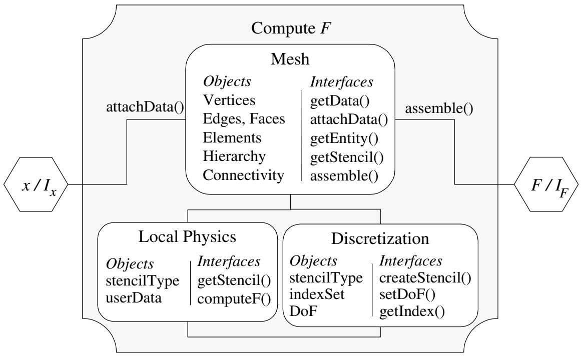

Figure 2: Schematic diagram of a nonlinear function evaluation.

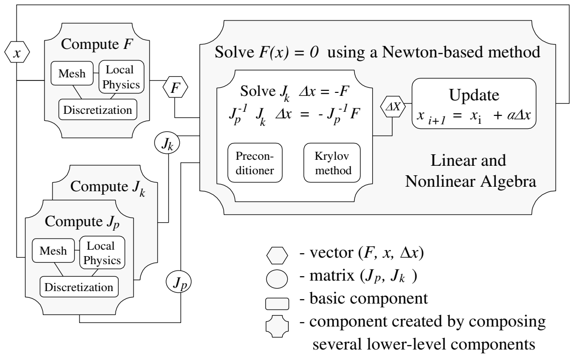

Figure 3: Schematic diagram of a Newton-type method.

ity (e.g., the standard 7-point stencil used in central finite differences or the vertices of an element used in finite element discretizations). This stencil can be accessed by both the mesh and the local physics components to obtain the spatial information and application-specific data necessary to compute F_i. Once the local values, F_i, have been computed, they can be assembled into the global vector by the mesh component, which simultaneously creates F's mapping, I_F.

An ongoing project at Argonne is exploring design issues for a general mesh management component that can interact with solver and visualization tools [10]. Key issues under consideration include the set of "atomic" methods for mesh manipulations, recognizing that aggregation is crucial for large-scale problems, and the trade-offs in performance versus degree of canonicalization.

3.2.2 Newton Numerics

Figure 3 illustrates the use of numerical components at various levels of abstraction for solving the nonlinear system $F(x) = 0$ with Newton-based methods. We first consider the solution of the linearized system, $F'(x)dx = -F(x)$, which is a key kernel within the overall simulation. Most modern linear iterative solvers may be viewed as the combination of a preconditioner (simple stationary iterative solver) and a Krylov subspace accelerator. Figure 3 illustrates these relationships within a component for linear solution, which in turn can be considered as part of a higher-level component for nonlinear solution.

The linear solver component requires input of a discretized linear operator (or matrix), $J_k \approx F'(x)$, and a vector F to define the algebraic system. Optionally, a different matrix, $J_p \approx F'(x)$, can be provided for use in building a preconditioner; we often choose J_p to be a relatively cheap approximation of the Jacobian (e.g., lower order discretization than F and J_k).

We can easily explore different algorithmic combinations and parallel data representations (e.g., sparse matrix storage schemes, matrix-free methods), since our interfaces exploit mathematical abstraction (see, e.g., [4]). Also, the ALICE microkernel's support for runtime binding of different toolkits facilitates experiments with functionality provided by various external solvers.

3.2.3 Jacobian Evaluation

The solution of nonlinear systems using Newton-type methods requires computation of a Jacobian matrix $F'(x)$ or, in the case of a matrix-free method, a Jacobian-vector product $F'(x) \cdot v$. While these derivatives can be approximated with finite differences, determining an appropriate differencing parameter can be difficult, and even reasonable choices may degrade algorithmic convergence in comparison to analytic code. Since code for computing derivatives analytically is often complicated and difficult to program by hand, we consider the use of automatic differentiation (AD), a technology for augmenting a computer program with statements for the computation of derivatives [11]. The AD process is not entirely automatic, as the user of an AD tool must identify a model's independent and dependent variables and perform some initialization.

Combining automatic differentiation technology with component-based software development can be mutually beneficial, and we have begun integrating AD technology with the nonlinear solver component of PETSc. Given a component for computing a function, $F(x)$, AD can generate a component for computing the Jacobian $F'(x)$ or a Jacobian-vector product $F'(x) \cdot v$. Furthermore, because component-based software has well-defined interfaces, the process of generating derivative code can be completely automated. While the current implementation is manual, there are no fundamental impediments to automating the process, and we intend to do so in the future.

We demonstrate the utility of these abstract component interfaces through some algorithmic experiments in a nonlinear PDE-based simulation. The application discussed here is three-dimensional compressible Euler flow arising in the context of aerodynamics [12], though the same numerical techniques are also under consideration for the Raleigh-Taylor simulations discussed in Section 2. This application uses nonlinear solvers within PETSc, which are organized according to the schematic diagram of Figure 3.

In particular, we compared both finite differencing and automatic differentiation for computing Jacobian-vector products, $J_k \cdot v$, in matrix-free Newton-Krylov methods. Figure 4 demonstrates some of the convergence benefits of using the automatic differentiation tool ADIFOR [6], where we see that AD overcomes challenges in selection of an appropriate differencing parameter for the finite difference case. This particular simulation, which used four processors of an IBM SP for a problem with 158,760 unknowns, solved the linearized Newton systems with restarted GMRES in conjunction with Restricted Additive Schwarz preconditioning. The preconditioner was built using J_p, which in this case was a finite difference approximation of $F'(x)$ that was held fixed over several nonlinear iterations.

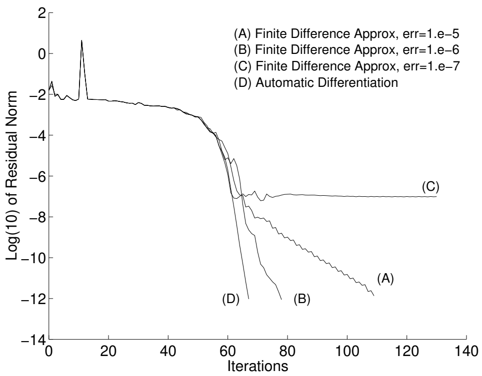

Figure 4: Convergence comparison of automatic differentiation and finite difference approximations in a matrix-free Newton method for compressible Euler flow.

4 Future Work

The infrastructure and interfaces presented in this paper represent some of the lines of research being pursued within the ALICE project. We have also begun to develop derivative-enhanced components [14], which can play an important role in sensitivity analysis, optimization, and inverse problems. In all of these activities, the fundamental objective is to provide a flexible environment for managing the complex interactions among a variety of numerical components, while maintaining the performance that scientific/engineering application developers expect and require.

Acknowledgments

We thank Bruce Fryxell, Kevin Olson, and Mike Singer for the images of the Raleigh-Taylor phenomenon presented in Section 1. We have developed the ideas presented in this paper from research and discussion with collaborators within the ALICE project and the Center on Astrophysical Thermonuclear Flashes. We thank all of these individuals, as well as members of the DOE-wide Common Component Architecture (CCA) Forum [2], for stimulating discussions of issues in component design for high-performance scientific computing.

References

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

[2] R. Armstrong et al., *Common Component Architecture Forum Web page.* http://www.acl.lanl.gov/cca-forum.

[3] I. Ba and B. Smith, *ALICE Memory Snooper Web page.* http://www.mcs.anl.gov/ams/, Mathematics and Computer Science Division, Argonne National Laboratory.

[4] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, *Efficient management of parallelism in object oriented numerical software libraries*, in Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhauser Press, 1997, pp. 163–202.

[5] ——, *A microkernel design for component-based parallel numerical software systems*, in Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, Oct 1998. (to appear).

[6] C. Bischof, A. Carle, P. Khademi, and A. Mauer, *ADIFOR 2.0: Automatic differentiation of Fortran 77 programs*, IEEE Computational Science & Engineering, 3 (1996), pp. 18–32.

[7] R. Clay et al., *Equation Solver Interface Forum Web page.* http://z.ca.sandia.gov/esi.

[8] A. Cleary, S. Kohn, S. Smith, and B. Smolinski, *Language interoperability mechanisms for high-performance scientific applications*, in Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, SIAM, Oct 1998. (to appear).

[9] L. Freitag, M. Jones, C. Ollivier-Gooch, and P. Plassmann, *SUMAA3d Web page.* http://www.mcs.anl.gov/sumaa3d/, Mathematics and Computer Science Division, Argonne National Laboratory.

[10] L. Freitag, M. Jones, and P. Plassmann, *Component integration for unstructured mesh algorithms and software*, in Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, SIAM, Oct 1998. (to appear).

[11] A. Griewank and S. Reese, *On the calculation of Jacobian matrices by the Markowitz rule*, in Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. F. Corliss, eds., SIAM, Philadelphia, 1991, pp. 126–135.

[12] W. D. Gropp, D. E. Keyes, L. C. McInnes, and M. D. Tidriri, *Globalized Newton-Krylov-Schwarz algorithms and software for parallel implicit CFD*, Tech. Rep. 98-24, ICASE, Aug. 1998. (to appear in Int. J. Supercomputer Applications).

[13] A. Hindmarsh et al., *PVODE Web page.* http://www.llnl.gov/CASC/PVODE, Lawrence Livermore National Laboratory.

[14] P. Hovland, B. Norris, and B. Smith, *Developing a derivative-enhanced object-oriented toolkit for scientific computations*, in Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, SIAM, Oct 1998. (to appear).

[15] *PSEware Web page.* http://www.extreme.-indiana.edu/pseware/, Indiana University.

[16] R. Rosner et al., *University of Chicago Center on Astrophysical Thermonuclear Flashes Web page.* http://www.asci.uchicago.edu.

[17] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, ACM Press, New York, 1997.

[18] S. Weerawarana, E. N. Houstis, J. R. Rice, A. C. Catlin, C. Crabill, C. C. Chui, and S. Markus, *PDELab: An object-oriented framework for building problem solving environments for PDE based applications*, Tech. Rep. CSD-TR-94-021, Department of Computer Sciences, Purdue University, 1994.