Parallel PDE-Based Simulations Using the Common Component Architecture

Lois Curfman McInnes[1], Benjamin A. Allan[2], Robert Armstrong[2], Steven J. Benson[1], David E. Bernholdt[3], Tamara L. Dahlgren[4], Lori Freitag Diachin[4], Manojkumar Krishnan[5], James A. Kohl[3], J. Walter Larson[1], Sophia Lefantzi[6], Jarek Nieplocha[5], Boyana Norris[1], Steven G. Parker[7], Jaideep Ray[8], and Shujia Zhou[9]

[1] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, `[mcinnes,benson,larson,norris]@mcs.anl.gov`
[2] Scalable Computing R & D, Sandia National Laboratories (SNL), Livermore, CA, `[baallan,rob]@ca.sandia.gov`
[3] Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, `[bernholdtde,kohlja]@ornl.gov`
[4] Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA, `[dahlgren1,diachin2]@llnl.gov`
[5] Computational Sciences and Mathematics, Pacific Northwest National Laboratory, Richland, WA, `[manojkumar.krishnan,jarek.nieplocha]@pnl.gov`
[6] Reacting Flow Research, SNL, Livermore, CA, `slefant@ca.sandia.gov`
[7] Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, UT, `sparker@cs.utah.edu`
[8] Advanced Software R & D, SNL, Livermore, CA, `jairay@ca.sandia.gov`
[9] Northrop Grumman Corporation, Information Technology Sector, Chantilly, VA, `szhou@pop900.gsfc.nasa.gov`

Summary. The complexity of parallel PDE-based simulations continues to increase as multimodel, multiphysics, and multi-institutional projects become widespread. A goal of component-based software engineering in such large-scale simulations is to help manage this complexity by enabling better interoperability among various codes that have been independently developed by different groups. The Common Component Architecture (CCA) Forum is defining a component architecture specification to address the challenges of high-performance scientific computing. In addition, several execution frameworks, supporting infrastructure, and general-purpose components are being developed. Furthermore, this group is collaborating with others in the high-performance computing community to design suites of domain-specific component interface specifications and underlying implementations.

This chapter discusses recent work on leveraging these CCA efforts in parallel PDE-based simulations involving accelerator design, climate modeling, combustion, and accidental fires and explosions. We explain how component technology helps to address the different challenges posed by each of these applications, and we highlight how component interfaces built on existing parallel toolkits facilitate the reuse of software for parallel mesh manipulation, discretization, linear algebra, integration, optimization, and parallel data redistribution. We also present performance data to demonstrate the suitability of this approach, and we discuss strategies for applying component technologies to both new and existing applications.

1 Introduction

The complexity of parallel simulations based on partial differential equations (PDEs) continues to increase as multimodel, multiphysics, multidisciplinary, and multi-institutional projects are becoming widespread. Coupling models and different types of science increases the complexity of the simulation codes. Collaboration across disciplines and institutions, while increasingly necessary, introduces new social intricacies into the software development process, such as different programming styles and different ways of thinking about problems. Added to these challenges, the software must cope with the multilevel memory hierarchies common to modern parallel computers where there may be three to five levels of data locality.

These challenges make it clear that the high-performance scientific computing community needs an approach to software development for parallel PDEs that facilitates managing such complexity while maintaining scalable and efficient parallel performance. Rather than being overwhelmed by the tedious details of parallel computing, computational scientists must be able to focus on the particular part of a simulation that is of primary interest to them (e.g., the physics of combustion) and employ well-tested and optimized code developed by experts in other facets of a simulation (e.g., parallel linear algebra and visualization). Traditional approaches, such as the widespread use of software libraries, have historically been valuable, but these approaches are being severely strained by this new complexity.

One goal of component-based software engineering (CBSE) is to enable interoperability among software modules that have been developed independently by different groups. CBSE treats applications as assemblies of software *components* that interact with each other only through well-defined *interfaces* within a particular execution environment, or *framework*. Components are a logical means of encapsulating knowledge from one scientific domain for use by those in others, thereby facilitating multidisciplinary interactions. The complexity of a given simulation is decomposed into bite-sized components that one or a few investigators can develop independently, thus enabling the collaboration of scores of researchers in the development of a single simulation. The glue that binds the components together is a set of common, agreed-upon interfaces. Multiple component implementations conforming to the same external interface standard should be interoperable, while providing flexibility to accommodate different aspects such as algorithms, performance characteristics, and coding styles. At the same time, the use of common interfaces facilitates the reuse of components across multiple applications. Even though details differ widely, many PDE-based simulations share the same overall software structure. Such applications could employ similar sets of components, which might conform to many of the same interfaces but differ in implementation details. This kind of software reuse enables the cross-pollination of both components and concepts across applications, projects, and problem domains.

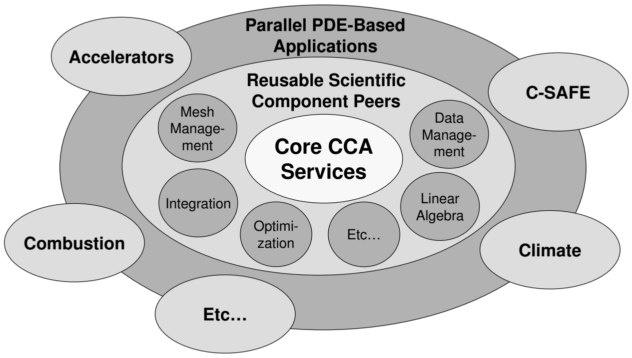

Fig. 1. Complete parallel PDE-based applications can be built by combining reusable scientific components with application-specific components; both can employ core CCA services to manage inter-component interactions.

The Common Component Architecture (CCA) [8,21,28] is designed specifically for the needs of parallel, scientific high-performance computing (HPC) in response to limitations in the general HPC domain of other, more widely used component approaches (see Section 3). The general-purpose design of the CCA is intended for use in a wide range of scientific domains for both PDE-based and non-PDE-based simulations.

As depicted in Figure 1, complete parallel PDE-based applications can be built in a CCA environment by combining various reusable scientific components with application-specific components. In keeping with the emphasis of this book, we explain (1) how component software can help manage the complexity of PDE-based simulations and (2) how the CCA, in particular, facilitates parallel scientific computations. We do this in the context of four motivating PDE-based application areas, which are introduced in Section 2. After presenting the basic concepts of the CCA in Section 3, we provide an overview of some reusable scientific components and explain how component interfaces built on existing parallel toolkits facilitate the reuse of software for parallel mesh manipulation, discretization, linear algebra, integration, optimization, and parallel data redistribution. Section 5 discusses strategies for applying component technologies to both new and existing applications, with an emphasis on approaches for the decomposition of PDE-based problems, including considerations for how to move from particular implementations to more general abstractions. Section 6 integrates these ideas through case studies that illustrate the application of component technologies and reusable components in the four motivating applications. Section 7 discusses conclusions and areas of future work.

2 Motivating Parallel PDE-Based Simulations

This section introduces four PDE-based application areas that motivate our work: accelerator design, climate modeling, combustion, and accidental fires and explosions.

2.1 Accelerator Modeling

Accelerators produce high-energy, high-speed beams of charged subatomic particles for research in high-energy and nuclear physics, synchrotron radiation research, medical therapies, and industrial applications. The design of next-generation accelerator facilities, such as the Positron-Electron Project (PEP)-II and Rare Isotope Accelerator (RIA), relies heavily on a suite of software tools that can be used to simulate many different accelerator experiments. Two of the codes used by accelerator scientists at the Stanford Linear Accelerator Center (SLAC) are Omega3P [120] and Tau3P [137]. Omega3P is an extensible, parallel, finite element-based code for the eigenmode modeling of large, complex three-dimensional electromagnetic structures in the frequency domain, while Tau3P provides solutions to electromagnetics problems in the time domain. Both codes make extensive use of unstructured mesh infrastructures to accommodate the complex geometries associated with accelerator models. In order to overcome barriers to computation and to improve functionality, both codes are being evaluated for possible extension.

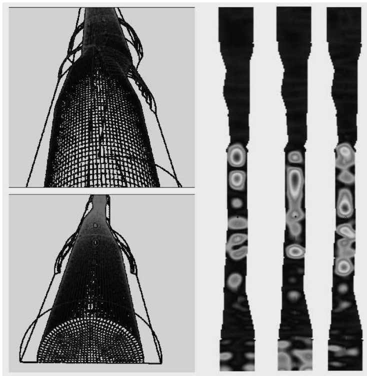

Fig. 2. State-of-the-art simulation tools are used to help design the next generation of accelerator facilities. *(Left)*: Mesh generated for the PEP-II interaction region using the CUBIT mesh generation package. Image courtesy of Tim Tautges of Sandia National Laboratories. *(Right)*: Excited fields computed using Tau3P. Image courtesy of the numerics team at SLAC.

For Tau3P, different discretization strategies are being explored to address long-time instabilities on certain types of meshes. Tau3P is based on a modified Yee algorithm formulated on an unstructured grid and uses a discrete surface integral (DSI) method to solve Maxwell's equations. Since the DSI scheme is known to have potential instabilities on nonorthogonal meshes, scientists are using a time filtering technique that maintains stability in most cases, but at a significantly higher computational cost. Unfortunately, integrating new discretization techniques is costly; and, because of resource constraints, several potentially useful methods cannot be investigated. A component-based approach that allows scientists to easily prototype different discretization and meshing strategies in a plug-and-play fashion would be useful in overcoming this obstacle.

For Omega3P, solutions are being explored that yield more accurate results without increasing the computational cost. That is, scientists are satisfied with the finite-element-based solver but cannot increase mesh resolution to reduce the large errors that occur in small regions of the computational domain. To overcome this barrier, SLAC scientists are working with researchers at Rensellaer Polytechnic Institute (RPI) to develop an adaptive mesh refinement (AMR) capability. Despite initially using a file-based information transfer mechanism, this effort has clearly demonstrated the advantage of AMR techniques to compute significantly more accurate solutions at a reduced computational cost. As described in Section 6.1, current efforts are centered on directly deploying these advanced capabilities in the Omega3P code by using a component approach. This approach has made the endeavor more tractable and has given scientists the flexibility of later experimenting with different underlying AMR infrastructures at little additional cost.

In order to facilitate the use of different discretization and meshing strategies, there is a need for a set of common interfaces that provide access to mesh and geometry information. A community effort to specify such interfaces is described in Section 4.1, and results of a performance study using a subset of those interfaces are discussed in Section 4.8.

2.2 Climate Modeling

Climate is the overall product of the mutual interaction of the Earth's atmosphere, oceans, biosphere, and cryosphere. These systems interact by exchanging energy, momentum, moisture, chemical fluxes, etc. The inherent nonlinearity of each subsystem's equations of evolution makes direct modeling of the *climate*—which is the set of statistical moments sampled over a large time scale—almost impossible. Instead, climate modeling is accomplished through integrations of coupled climate system models for extended periods, ranging from the century to millennial time scales, logging of model history output sampled at short time scales, and subsequent off-line analysis to compute climate statistics.

PDEs arise in many places in the climate system, most significantly in the dynamics of the atmosphere, ocean, and sea-ice. The ocean and atmosphere are both modeled as thin spherical shells of fluid in a rotating reference frame, using in each case a system of coupled PDEs governing mass, energy, and momentum conservation, called *the primitive equations*. Modern sea-ice models simulate the formation and melting of ice (the *thermodynamics* of the problem), how the ice pack is forced by surface winds and ocean currents, and how it behaves as a material (its *dynamics* and *rheology*). Schemes such as the elastic-viscous plastic (EVP) scheme [60] involve the solution of PDEs.

Climate modeling is a grand challenge high-performance computing application, requiring highly efficient and scalable algorithms capable of providing the high throughput needed for long-term integrations. To illustrate the high simulation costs, we consider the NASA finite-volume General Circulation Model. A 500-model-day simulation using this model, with a horizontal resolution of $0.5°$ latitude by $0.625°$

longitude and 32 vertical layers, takes a wall-clock day to run on a 1.25-GHz Compaq AlphaServer SC45 with 250 CPUs [83].

The requirements for coping with multiple, coupled physical processes as well the requirements for parallel computing make software development even more challenging. The traditional development process for these models was the creation of highly entangled applications that made little reuse of code and made the interchange of functional units difficult. In recent years, the climate/weather/ocean (CWO) community has embarked on an effort to increase modularity and interoperability, the main motivation being a desire to accelerate the development, testing, and validation cycle. This effort is positioning the community for the introduction of software component technology, and there is now an emerging community wide application framework, the Earth System Modeling Framework [67].

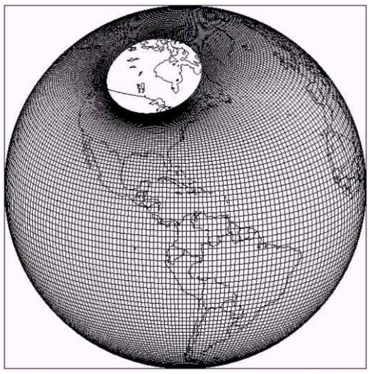

Fig. 3. Displaced pole grid on which the Parallel Ocean Program ocean model [104] solves its primitive equations. The polar region is displaced to lie over land, thereby minimizing the problems encountered at high latitudes by finite-difference schemes. Image courtesy of Phillip Jones and Richard Smith, Los Alamos National Laboratory.

In Section 4.7 we discuss the use of CCA components for climate model coupling. In Section 6.2 we describe the multiple software scales at which component technology is appropriate in climate system models. We briefly describe the ESMF and its relationship to the CCA, and we provide an example of CWO code refactoring to to make it component friendly. We also describe a prototype component-based advection model that combines the interoperable component paradigms of the CCA and ESMF.

2.3 Combustion

The study of flames, experimentally and computationally, requires the resolution of a wide range of length and time scales arising from the interaction of chemistry, radiation, and transport (diffusive and convective). The complexity and expense involved in the experimental study of flames were recognized two decades ago, and the Combustion Research Facility [37] was created as a “user facility” whose equipment and expertise would be freely available to industry and academia. Today a similar challenge is being faced in the high-fidelity numerical simulations of flames [105]. Existing simulations employ a variety of numerical and parallel computing strategies to achieve an accurate resolution of physics and scales, with the unfortunate side effect of producing large, complex and ultimately unwieldy codes. Their lack of extensibility and difficulty of maintenance have been a serious impediment and were the prime motive for establishing in 2001 the Computational Facility for Reacting

Flow Science (CFRFS) [93], a “simulation facility” where various numerical algorithms, physical and chemical models, meshing strategies, and domain partitioners may be tested in flame simulations.

In the CFRFS project, flames are solved by using the low Mach number form of the Navier-Stokes equation [92, 133], augmented by evolution equations for the various chemical species and an energy equation with a source term to incorporate the contribution from chemical reactions. The objective of the project is to simulate laboratory-sized flames with detailed chemistry, a problem that exhibits a wide spectrum of length and time scales. Block-structured adaptive meshes [18] are used to limit fine meshes only where (and when) required; operator-splitting [68, 117] is used to treat stiff chemical terms implicitly in time, while the convective and diffusive terms are advanced explicitly. In many cases, the stiff chemical system can be rendered nonstiff (without any appreciable loss of fidelity) by projection onto a lower-dimensional manifold. The identification of this manifold and the projection onto it are achieved by computational singular perturbation (CSP) [70, 75], a multiscale asymptotic method that holds the promise of significantly reducing the cost of solving the chemical system.

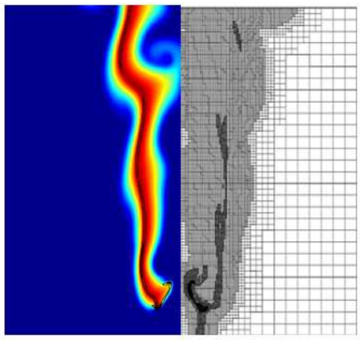

Fig. 4. A 10-cm-high pulsating methane-air jet flame, computed on an adaptive mesh. On the left is the temperature field with a black contour showing regions of high heat release rates. On the right is the adaptive mesh, in which regions corresponding to the jet shear layer are refined the most.

Given the scope of the simulation facility, the requisite degree of flexibility and extensibility clearly could not be achieved without a large degree of modularization and without liberating the users (with widely varying levels of computational expertise) from the strait jacket imposed by global data-structures and models. Modularization was achieved by adopting a component-based architecture, and the multidimensional Fortran array was adopted as the basic unit of data exchange among scientific components. The simulation facility can thus be viewed as a toolkit of components, each embodying a certain numerical or physical functionality, mostly implemented in Fortran 77, with thin C++ “wrappers” around them.

In Section 4.2 we discuss SAMR components used in this application, and in Section 5 we detail the strategy we adopted to decompose mathematical and simulation requirements into modules, while preserving a close correspondence between the software components and identifiable physics in the governing equations. In Section 6.3 we demonstrate the payoffs of adopting such a *physics-based* approach.

2.4 Accidental Fires and Explosions

In 1997 the University of Utah created an alliance with the U.S. Department of Energy (DOE) Accelerated Strategic Computing Initiative (ASCI) to form the Center for the Simulation of Accidental Fires and Explosions (C-SAFE) [55]. C-SAFE focuses on providing state-of-the-art, science-based tools for the numerical simulation of accidental fires and explosions, especially within the context of handling and storing highly flammable materials. The primary objective of C-SAFE is to provide a software system in which fundamental chemistry and engineering physics are fully coupled with nonlinear solvers, optimization, computational steering, visualization, and experimental data verification, thereby integrating expertise from a wide variety of disciplines. Simulations using this system will help to better evaluate the risks and safety issues associated with fires and explosions in accidents involving both hydrocarbon and energetic materials. A typical C-SAFE problem is shown in Figure 5. Section 6.4 discusses the use of component concepts in this application and demonstrates scalable performance on a variety of parallel architectures.

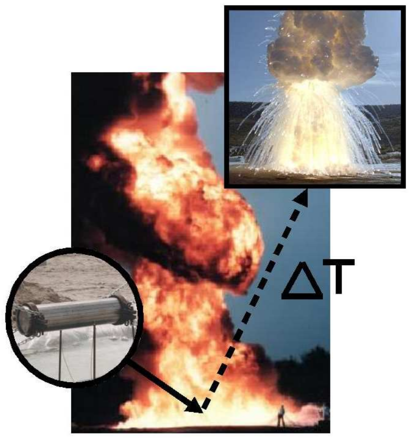

Fig. 5. A typical C-SAFE problem involving hydrocarbon fires and explosions of energetic materials. This simulation involves fluid dynamics, structural mechanics, and chemical reactions in both the flame and the explosive. Accurate simulations of these events can lead to a better understanding of high-energy explosives, can help evaluate the design of shipping and storage containers for these materials, and can help officials determine a response to various accident scenarios. The fire image is courtesy of Schonbucher Institut for Technische Chemie I der Universitat Stuttgart, and the images of the container and explosion are courtesy of Eric Eddings of the University of Utah.

3 High-Performance Components

High-performance components offer a means to deal with the ever-increasing complexity of scientific software, including the four applications introduced in Section 2. We first introduce general component concepts, discuss the Common Component Architecture (CCA), and then introduce two simple PDE-based examples to help illustrate CCA principles and components.

3.1 Component-Based Software Engineering

In addition to the advantages of component-based software engineering (CBSE; see, e.g., [121]) discussed in Section 1, component-based approaches offer additional benefits, including the following:

- *Plug-and-play assembly* improves productivity, especially when a significant number of components can be used without customization, and simplifies the evolution of applications to meet new requirements or address new problems.
- *Clear interfaces and boundaries* around components simplify the composition of multiple componentized libraries in ways that may be difficult or impossible with software libraries in their traditional forms. This approach also helps researchers to focus on the particular aspects of the problem corresponding to their interests and expertise.
- *Components enable adaptation* of applications in ways that traditional design cannot. For example, interface standards facilitate swapping of components to modify behavior or performance; such changes can even be made automatically without user intervention [96].

As implied above and in Section 1, CBSE can be thought of, in many respects, as an extension and refinement of the use of software libraries—a popular and effective approach in modern scientific computing. Components are also related to "domain-specific computational frameworks" or "application frameworks," which have become popular in recent years (e.g., Cactus [6], ESMF [67], and PRISM [53]). Typically, such environments provide deep computational support for applications in a given domain, and applications are constructed at a relatively high level. Many application frameworks even have a componentlike structure at the high level, allowing arbitrary code to be plugged in to the framework. Application frameworks are more constrained than general component environments because the ability to reuse components across scientific domains is quite limited, and the framework tends to embody assumptions about the workflow of the problem domain. General component models do not impose such constraints or assumptions and provide broader opportunities for reuse. Domain-specific frameworks can be constructed within general component environments by casting the domain-specific infrastructure and workflow as components.

A number of component models have attained widespread use in mainstream computing, especially Enterprise JavaBeans [44,112], Microsoft's COM/DCOM [26, 88, 89], and the Object Management Group's CORBA and the CORBA Component Model [97]. Despite its advantages, however, CBSE has found only limited adoption in the scientific computing community to date [64, 84, 102]. Unfortunately, the commodity component models tend to emphasize distributed computing while more or less ignoring parallel computing, impose significant performance overheads, or require significant changes to existing code to enable it to operate within the component environment. Additional concerns with many component models include support for programming languages important to scientific computing, such as Fortran; support for data types, such as complex numbers and arrays; and operating system support. The Common Component Architecture has been developed in direct response to the need for a component environment targeted to the needs of high-performance scientific computing.

3.2 The Common Component Architecture

The Common Component Architecture [28] is the core of an extensive research and development program focused on understanding how best to utilize and implement component-based software engineering practices in the high-performance scientific computing area, and on developing the specifications and tools that will lead to a broad spectrum of CCA-based scientific applications. A comprehensive description of the CCA, including more detailed presentations of many aspects of the environment is available [21]; here we present a brief overview of the CCA environment, focusing on the aspects most relevant to parallel PDE-based simulations.

The specification of the Common Component Architecture [29] defines the rights, responsibilities, and relationships among the various elements of the model. Briefly, the elements of the CCA model are as follows:

- *Components* are units of software functionality that can be composed together to form applications. Components encapsulate much of the complexity of the software inside a black box and expose only well-defined interfaces.
- *Ports* are the abstract interfaces through which components interact. Specifically, CCA ports provide procedural interfaces that can be thought of as a class or an interface in object-oriented languages, or a collection of subroutines, or a module in a language such as Fortran 90. Components may provide ports, meaning that they implement the functionality expressed in a port (called *provides ports*), or they may use ports, meaning that they make calls on a port provided by another component (called *uses ports*). The notion of CCA ports is less restrictive than hardware ports: ports are not assumed to be persistent, e.g., available throughout an application's lifetime, and each port can have different access attributes, such as the number of simultaneous connections.
- *Frameworks* manage CCA components as they are assembled into applications and executed. The framework is responsible for connecting *uses* and *provides* ports without exposing the components' implementation details. The framework also provides a small set of standard services that are available to all components.

Several frameworks that implement the CCA specification and support various computing environments have been developed. Ccaffeine [3] and SCIRun2 [138], used by the applications in this chapter, focus on high-performance parallel computing, while XCAT [52, 61] primarily supports distributed computing applications; several other frameworks are being used as research tools.

The importance of efficient and scalable performance in scientific computing is reflected in both the design of the CCA specification and the features of the various framework implementations. The CCA's *uses/provides* design pattern allows components in the same process address space to be invoked directly, without intervention by the framework, and with data passed by reference if desired (also referred to as "direct connect," "in-process," or "co-located" components). In most CCA frameworks, this approach makes local method calls between components equivalent to C++ virtual function calls, an overhead of roughly 50 ns on a 500 MHz Pentium system (compared to 17 ns for a subroutine call in a non-object-oriented language such as C or Fortran) [20].

The CCA approach to parallelism. For parallel computing, the CCA has chosen not to specify a particular parallel programming model but rather to allow framework and application developers to use the programming models they prefer. This approach has several advantages, the most significant of which is that it allows component developers to use the model that best suits their needs, greatly facilitating the incorporation of existing parallel software into the CCA environment. Figure 6 shows schematically a typical configuration for a component-based parallel application in the Ccaffeine framework. For a single-program multiple-data (SPMD) application, each parallel process would be loaded with the same set of components, with their ports connected in the same way. Interactions *within a given parallel process* occur through normal CCA mechanisms, getting and releasing ports on other components and invoking methods on them. These would generally use the local direct connect approach mentioned above, to minimize the CCA-related overhead. Interactions within the parallel cohort of a given component are free to use the parallel programming model they prefer, for example MPI [91], PVM [48], or Global Arrays [94, 98]. Different sets of components may even use different programming models, an approach that facilitates the assembly of applications from components derived from software developed for different programming models. This approach imposes no CCA-specific overhead on the application's parallel performance. Such mixing of programming models can occur for components that interact at relatively coarse grained levels with loose coupling (for example, two parts of a multi-model physics application, such as the climate models discussed in Section 6.2). In contrast, sets of relatively fine grain and tightly coupled components (for example, the mesh and discretization components shown in Figure 7) must employ compatible parallel programming models. Multiple-program multiple-data (MPMD) applications are also supported through a straightforward generalization of the SPMD model. It is also possible for a particular CCA framework implementation to provide its own parallel programming model, as is the case with the Uintah framework discussed in Section 6.4.

Language interoperability. A feature of many component models, including the CCA, is that components may be composed together to form applications regardless of the programming language in which they have been implemented. The CCA provides this capability through the Scientific Interface Definition Language (SIDL) [38], which component developers can employ to express component interfaces. SIDL works in conjunction with the Babel language interoperability tool [38, 74], which currently supports C, C++, Fortran 77, Fortran 90/95, and Python, with work under way on Java. SIDL files are processed by the Babel compiler, which generates the glue code necessary to enable the caller and callee to be in any supported language. The generated glue code handles the translation of arguments and method calls between languages. Babel also provides an object-oriented

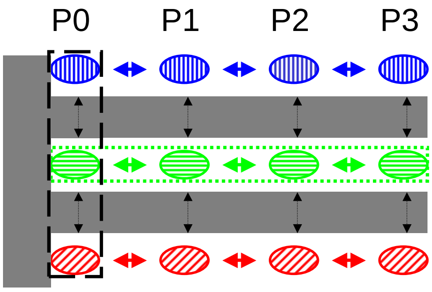

Fig. 6. A schematic representation of the CCA parallel programming environment in the single component/multiple data (SCMD) paradigm. Parallel processes, labeled P0,...,P3, are loaded with the same set of three components. Components in the same process (vertical dashed box) interact using standard CCA port-based mechanisms, while parallel components of the same type (horizontal dotted box) interact using their preferred parallel programming model.

(OO) model, which can be used even in non-OO languages such as C and Fortran. On the other hand, neither Babel nor the CCA requires that interfaces be strongly object-oriented; such design decisions are left to the component and interface designers.

The developers of Babel are also sensitive to concerns about performance. Where Babel must translate arguments for method calls (because of differing representations in the underlying languages), there will clearly be some performance penalty. Since most numerical types do not require translation, however, in many cases Babel can provide language interoperability with no additional performance cost [20]. In general, the best strategy is for designers and developers to be aware of translation costs, and take them into account when designing interfaces, so that wherever possible enough work is done within the methods so that the translation costs are amortized; see Section 4.8 for performance overhead studies.

Incorporating components. The CCA employs a minimalist design philosophy to simplify the task of incorporating existing software into the CCA environment. Generally, as discussed in Section 3.3, one needs to add to an existing software module just a single method that informs the framework which ports the component will provide for use by other components and which ports it expects to use from others. Within a component, calls to ports on other components may have slightly different syntax, and calls must be inserted to obtain and release the handle for the port. Experience has shown that componentization of existing software in the CCA environment is straightforward when starting from well-organized code [5, 79, 95]. Moreover, the componentization can be done incrementally, starting with a coarse-grained decomposition (possibly even an entire simulation, if the goal is coupled simulations) and successively refining the decomposition when opportunities arise to replace functionality with a better-performing component.

Common interfaces. Interfaces are clearly a key element of the CCA and of the general concept of component-based software engineering; they are central to the interoperability and reuse of components. We note that except for a very small number of interfaces in the CCA specification, typically associated with framework services, the CCA does *not* dictate "standard" interfaces—application and component developers are free to define and use whatever interfaces work best for their purposes.

However, we do strongly encourage groups of domain experts to work together to develop interfaces that can be used across a variety of components and applications. Numerous such efforts are under way, including mesh management, linear algebra, and parallel data redistribution, all of which are related to the applications described in this chapter and are discussed in Section 4. Anyone interested in these efforts, or in launching other standardization efforts, is encouraged to contact the authors.

3.3 Simple PDE Examples

We next introduce two simple PDE examples to help illustrate CCA principles and components. While we have deliberately chosen these examples to be relatively simple and thus straightforward to explain, they incorporate numerical kernels and phases of solution that commonly arise in the more complicated scientific applications that motivate our work, as introduced in Section 2.

Steady-State PDE Example

The first example is Laplace's equation on a two-dimensional rectangular domain: $\nabla^2\phi(x,y) = 0, \;\; x \in [0,1], \;\; y \in [0,1]$, with $\phi(0,y) = 0$, $\phi(1,y) = \sin(2\pi y)$, and $\frac{\partial\phi}{\partial y}(x,0) = \frac{\partial\phi}{\partial y}(x,1) = 0$. This system can be discretized by using a number of different methods, including finite difference, finite element, and finite volume techniques on either a structured or an unstructured mesh. This example has characteristics of the large, sparse linear systems that are at the heart of many scientific simulations, yet it is sufficiently compact to enable the demonstration of CCA concepts and code.

The composition of this CCA application is shown by a component wiring diagram in the upper portion of Figure 7; the graphical interface of the Ccaffeine [3] framework enables similar displays of component interactions. This example employs components (as represented by large gray boxes) for unstructured mesh management, discretization, and linear solution, which are further discussed in Section 4, as well as an application-specific driver component, which is discussed below. The lines in the diagram between components represent connections between *uses* and *provides* ports, which are denoted by rectangular boxes that are white and checkered, respectively. For example, the discretization component's "Mesh" *uses* port is connected to the unstructured mesh component's "Mesh" *provides* port, so that the discretization component can invoke the mesh interface methods that the mesh component has implemented. The special *GoPort* (named "Go" in this application driver) starts the execution of the application.

The application scientist's perspective. The application-specific driver component plays the role of a user-defined `main` program in traditional library-based applications. CCA frameworks do not require that an application contain a definition of a `main` subroutine. In fact, in many cases, `main` is not defined by the user; instead, a definition in the framework is used. In that case, a driver component partially fulfills the role of coordinating some of the application's components; the actual instantiation and port connections can be part of the driver as well, or these tasks can be accomplished via a user-defined script or through a graphical user interface. A CCA framework can support multiple levels of user control over component instantiation and connection; here we present only one of the higher levels, where the user takes advantage of a framework-supplied `main` program, as well as framework-specific concise mechanisms for application composition. In this example, the application could be composed by using a graphical user interface, such as that provided with the Ccaffeine [3] framework, by selecting and dragging component classes to instantiate them, and then clicking on pairs of corresponding ports to establish connections. Alternatively, the application could be composed with a user-defined script.

In addition to writing a driver component, typical application scientists would also write custom components for the other parts of the simulation that are of direct interest to their research, for example the discretization of a PDE model (see Section 6.3 for a discussion of the approach used by combustion researchers). These application-specific components can then be used in conjunction with external component-based libraries for other facets of the simulation, for example, unstructured mesh management (see Section 4.1) and linear solvers (see Section 4.4). As discussed in detail in Section 4.1, if multiple component implementations of a given functionality adhere to common port specifications, then different implementations, which have been independently developed by different groups, can be seamlessly substituted at runtime to facilitate experimentation with a variety of algorithms and data structures.

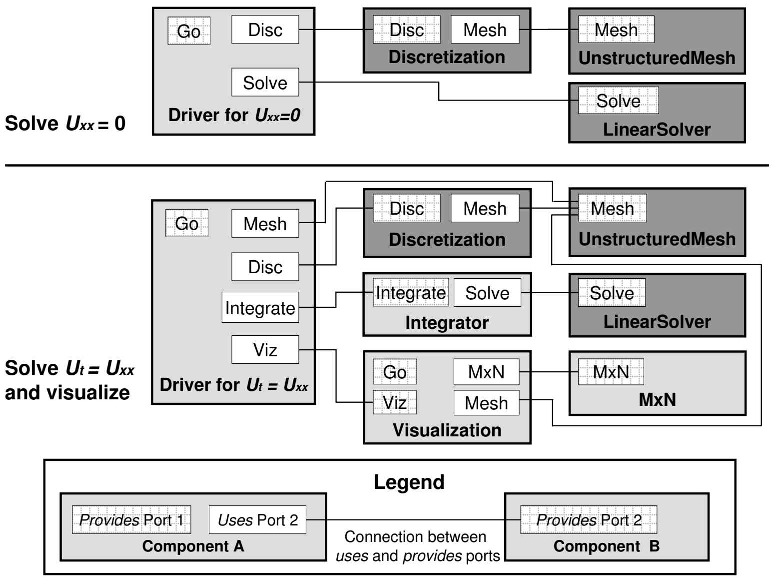

Fig. 7. Two component wiring diagrams for (*top*) a steady-state PDE example and (*bottom*) a time-dependent PDE example demonstrate the reuse of components for mesh management, discretization, and linear solvers in two different applications.

```
package laplace version 1.0 {
  class Driver implements gov.cca.Component,
                          gov.cca.ports.GoPort
  {
    // The only method required to be a CCA component.
    void setServices(in gov.cca.Services services);
    // The GoPort method that returns 0 if successful.
    int go();
  }
}
```

Fig. 8. SIDL definition of the driver component for the steady-state PDE example.

A closer look at the application-specific driver component. Figure 8 shows the SIDL definition of the driver component for the solution of the steady-state PDE example in Figure 7. As discussed in Section 3.2, the use of SIDL for the component interface enables the component to interact easily with other components that may be written in a variety of programming languages. The `Driver` SIDL class must implement the `setServices` and `go` methods, which are part of the `gov.cca.Component` and `gov.cca.ports.GoPort` interfaces, respectively [29]. For this example, we used Babel to generate a C++ implementation skeleton, to which we then added the application-specific implementation details, portions of which are discussed next.

Figure 9 shows the implementation of the `setServices` method, which is generally used by components to save a reference to the framework `Services` object and to register *provides* and *uses* ports with the framework. The `frameworkServices` user-defined data member in the `Driver_impl` class stores the reference to the services object, which can be used subsequently to obtain and release ports from the framework and for other services. To provide the “Go” port, the driver component's `self` data member (a Babel-generated reference similar to the `this` pointer in C++) is first cast as a `gov::cca::Port` in the assignment of `self` to `port`; then the `addProvidesPort` services method is used to register the *provides* port of type `gov.cca.ports.GoPort` with the framework, giving it the name “Go”. A `gov.cca.TypeMap` object, `tm`, is created and passed to each call that registers *provides* and *uses* ports; in larger applications, these name-value-type dictionaries can be used for storing problem and other application-specific parameters.

Figure 10 shows an abbreviated version of the `go` method implementation for the simple steady-state PDE example. First, we obtain a reference to the discretization port “Disc” from the framework services object, `frameworkServices`. Note that in Babel-generated C++ code, the casting of the `gov::cca::Port` object returned by `getPort` to type `disc::Discretization` is performed automatically. The discretization component uses finite elements to assemble the linear system in the implementation of the `createFESystem` method, which includes information exchange with the unstructured mesh component through the “Mesh” port. The linear system is then solved by invoking the `apply` method on the linear solver component. Finally, all ports obtained in the `go` method are released via the `releasePort` framework services method.

The linear algebra interfaces in this example are based on the TOPS solver interfaces [114] (also see Section 4.4). The matrix and vector objects in this example are not components themselves, but are created as regular objects by the driver component and then modified and used in the mesh, discretization, and solver components. While such linear algebra objects could be implemented as components themselves, we chose to use a slightly more lightweight approach (avoiding one layer of abstraction) because they have relatively fine-grain interfaces, e.g., setting individual vector and matrix elements. In contrast, the solver component, for example, provides a port whose methods perform enough computation to make the overhead of port-based method invocation negligible (see, e.g., [95]).

Time-Dependent PDE Example

The second PDE that we consider is the heat equation, given by $\frac{\partial \phi}{\partial t} = \nabla^2 \phi(x, y, t)$, $x \in [0, 1], \quad y \in [0, 1]$, with $\phi(0, y, t) = 0$, $\phi(1, y, t) = \frac{1}{2} \sin(2\pi y) \cos(t/2)$, $\frac{\partial \phi}{\partial y}(x, 0, t) = \frac{\partial \phi}{\partial y}(x, 1, t) = 0$. The initial condition is $\phi(x, y, 0) = \sin(\frac{1}{2}\pi x)$ $\sin(2\pi y)$. As shown by the component wiring diagram in the lower portion of Figure 7, this application reuses the unstructured mesh, discretization, and linear algebra components employed by the steady-state PDE example and introduces a time integration component as well as components for parallel data redistribution and visualization. These reusable scientific components are discussed in further detail in Section 4.

Another component-based solution of the heat equation, but on a *structured mesh*, can be found at [109]. This approach employs different discretization and mesh components from those discussed above but reuses the same integrator. This CCA example is freely downloadable from [109], including scripts for running the code.

```
void laplace::Driver_impl::setServices (
  /*in*/ ::gov::cca::Services services )
throw ( ::gov::cca::CCAException)
{
  // frameworkServices is a programmer-defined private data member of
  // the Driver_impl class, declared as
  //     ::gov::cca::Services frameworkServices
  // in the Babel-generated laplace_Driver_impl.hh file
  frameworkServices = services;

  // Provide a Go port; the following statement performs an implicit cast
  gov::cca::Port port = self;
  gov::cca::TypeMap tm = frameworkServices.createTypeMap();
  frameworkServices.addProvidesPort(port, "Go", "gov.cca.ports.GoPort",tm);

  // Use Discretization and Solver ports
  frameworkServices.registerUsesPort("Disc", "disc.Discretization",tm);
  frameworkServices.registerUsesPort("Solver", "solvers.LinearSolver",tm);
}
```

Fig. 9. Laplace application driver code fragment showing the C++ implementation of the `setServices` method of the `gov.cca.Component` interface.

More detailed CCA tutorial materials, including additional sample component codes as well as the Ccaffeine framework and Babel language interoperability tool, are available via `http://www.cca-forum.org/tutorials`. We recommend this site as a starting point for individuals who are considering the use of CCA tools and components.

These examples illustrate one of the ways that components can participate in a scientific application. In larger applications, such as those introduced in Section 2, different components are typically developed by different teams, often at different sites and times. Some of these components are thin wrappers over existing numerical libraries, while others are implemented from scratch to perform some application-specific computation, such as the discretization components in Figure 7. The CCA component model, like other component models, provides a specification and tools that facilitate the development of complex, multi-project, multi-institutional software. In addition to helping manage software development complexity, the simple port abstraction (1) enables the definition of explicit interaction points between parts

```
int32_t laplace::Driver_impl::go() throw () {

  disc::Discretization discPort;
  solvers::Solver linearSolverPort;
  try {
    // Get the discretization port.
    discPort = frameworkServices.getPort("Disc");

    // The layout object of type solvers::Layout_Rn is a data member
    // of the Driver_impl class describing how vector and matrix
    // data is laid out across processors; it also provides a factory
    // interface for creating parallel vectors and matrices.

    // Create the matrix, A, and right-hand-side vector, b
    solvers::Vector_Rn b = layout.createGlobalVectors(1)[0];
    solvers::Matrix_Rn A = layout.createOperator(layout);

    // Assemble A and b to define the linear system, Ax=b
    discPort.createFESystem(A, b);

    // Get the solver port
    linearSolverPort = frameworkServices.getPort("Solver");

    // Create the solution vector, x
    solvers::Vector_Rn x = layout.createGlobalVectors(1)[0];

    // Initialize and solve the linear system
    linearSolverPort.setOperator(A);
    linearSolverPort.apply(b, x);

    // Release ports
    frameworkServices.releasePort("Disc");
    frameworkServices.releasePort("Solver");
    return 0;
  } catch ( gov::cca::CCAException& e) { return -1; }
}
```

Fig. 10. Laplace application driver code fragment showing the C++ implementation of the `go` method from the `gov.cca.ports.GoPort` interface. Exceptions are converted to the function return code specified in Figure 8 with the try/catch mechanism.

of an application; (2) facilitates the use of thoroughly tuned external components implemented by experts; and (3) allows individual components to be developed, maintained, and extended independently, with minimal impact on the remainder of the application.

4 Reusable Scientific Components

Various scientific simulations often have similar mathematics and physics, but currently most are written in a stovepipe fashion by a small group of programmers with minimal code reuse. As demonstrated in part by Figure 7, a key advantage of component-oriented design is software reuse. Components affect reuse in two ways: (1) because an exported port interface is simpler to use than the underlying software, a component should be easier to import into a simulation than to rewrite from scratch; and (2) because *common* interfaces for particular functionalities can be employed by many component implementations, different implementations can be easily substituted for one another to enhance performance for a target machine, data layout, or parameter set. In Sections 4.1 through 4.7 we detail these two facets of reusability in terms of several current efforts to develop component implementations and domain-specific groups devoted to defining common interfaces for various numerical and parallel computing capabilities. In Section 4.8 we demonstrate that the overhead associated with CCA components is negligible when appropriate levels of abstraction are employed.

Component implementations can directly include the code for core numerical and parallel computing capabilities, and indeed new projects that start from scratch typically do so. However, many of the component implementations discussed in this section employ the alternative approach of providing thin wrappers layered on top of existing libraries, thereby offering optional new interfaces that make these independently developed packages easier to use in combination with one another in diverse projects. Section 5 discusses some of the issues that we have found useful to consider when building these component interfaces. The web site `http://www.cca-forum.org` has current information on the availability of these components as well as others.

4.1 Unstructured Mesh Management

Unstructured meshes are employed in many PDE-based models, including the accelerator application introduced in Section 2.1 and the simple examples discussed in Section 3.3. The Terascale Simulation Tools and Technologies (TSTT) Center [124], established in 2001, is developing common interface abstractions for managing mesh, geometry, and field data for the numerical solution of PDEs. As shown in Figure 11, the common TSTT mesh interface facilitates experimentation with different mesh management infrastructures by alleviating the need for scientists to write separate code to manage the interactions between an application and different meshing tools.

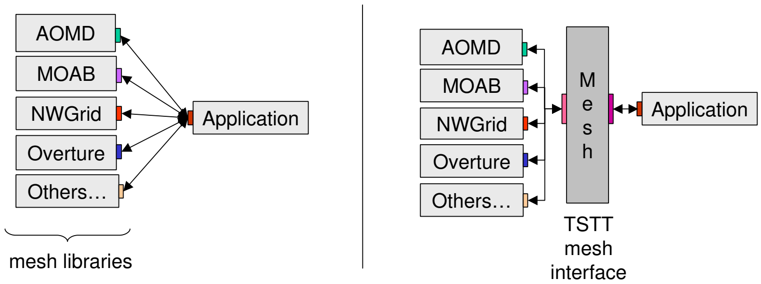

Fig. 11. *(Left)*: The current interface situation connecting an application to m mesh management systems through m different interfaces. Because developing these connections is often labor-intensive for application scientists, experimentation with various mesh systems is severely inhibited. *(Right)*: The desired interface situation, in which many mesh systems that provide similar functionality are compliant with a single common interface. Scientists can more easily explore the use of different systems without needing to commit to a particular solution strategy that could prematurely lock their application into a specific mesh management system.

TSTT Mesh Interfaces

The TSTT interfaces include *mesh data*, which provides the geometric and topological information associated with the discrete representation of a computational domain; *geometric data*, which provides a high-level description of the domain boundaries, for example, a CAD model; and *field data*, which provides the time-dependent physics variables associated with application solutions. The TSTT data model covers a broad spectrum of mesh types and functionalities, ranging from a nonoverlapping, connected set of entities (e.g., a finite element mesh) to a collection of such meshes that may or may not overlap to cover the computational domain. To date, TSTT efforts have focused on the development of mesh query and modification interfaces at varying levels of granularity. The basic building blocks for the TSTT interfaces are *mesh entities*, for example, vertices, edges, faces, and regions, and *entity sets*, which are arbitrary groupings of mesh entities that can be related hierarchically or by subsets. Functions have been defined that allow the user to access mesh entities using arrays or iterators, attach user-defined data through the use of tags, and manipulate entity sets using Boolean set operations.

The TSTT mesh interface is divided into several ports. The core interface provides basic functionality for loading and saving the mesh, obtaining global information such as the number of entities of a given type and topology, and accessing vertex coordinate information and adjacency information using both primitive arrays and opaque entity handle arrays. Additional ports are defined that provide single entity iterators, workset iterators, and mesh modification functionality. Figure 12 shows an example C-code client that uses the TSTT interface. The `mesh` variable represents a pointer to an object of type `TSTT_mesh`, which can be either an object created via a call to a Babel-generated constructor or a *uses* port provided by a mesh component instantiated in a CCA framework (for example, see Figure 7). The mesh data is loaded from a file whose name is specified via a string. Because many of the TSTT functions work on both the full mesh and on subsets of the mesh, the user must first obtain the root entity set using `getRootSet` to access the vertex and face information. In this example, the user asks for the handles associated with the triangular elements with the `getEntities` call. Once the triangle handles have been obtained, the user can access the adjacent vertices and their coordinate information either one handle at a time as shown in the example, or using a single function call that returns the adjacency information for all of the handles simultaneously.

```
#include "TSTT.h"

/* ... */
  void   *root_entity_set;
  void   **tri_handles, **adj_vtx_handles;
  int    i, num_tri, num_vtx, coords_size;
  double *coords;

  /* Load the data into the previously created mesh object */
  TSTT_mesh_load(mesh,''mesh.file'');

  /* Obtain a handle to the root entity set */
  root_entity_set = TSTT_mesh_getRootSet(mesh);

  /* Obtain handles to the triangular elements in the mesh */
  TSTT_mesh_getEntities(mesh,root_entity_set, TSTT_EntityType_FACE,
                    TSTT_EntityTopology_TRIANGLE, &tri_handles,
                    &num_tri);

  /* For each triangle, obtain the corner vertices and their
     coordinates */
  for (i=0;i<num_tri;i++) {
    TSTT_mesh_getEntAdj(mesh,tri_handle[i],TSTT_EntityType_VERTEX,
                    &adj_vtx_handles,&num_vtx);
    TSTT_mesh_getVtxArrCoords(mesh,adj_vtx_handles,num_vtx,
                    TSTT_StorageOrder_BLOCKED,&coords,&coords_size);
  }
/* ... */
```

Fig. 12. An example code fragment showing the use of the TSTT interface in C to load a mesh and retrieve the triangular faces and their corner vertex coordinates.

More information on the mesh interfaces and the TSTT Center can be found in [124]. Preliminary results of a performance study of the use of a subset of the mesh interfaces can be found in Section 4.8.

TSTT Mesh Component Implementations

Implementations of the TSTT mesh interfaces are under way at several institutions. The ports provided by these mesh components include the “Mesh” port, which gives basic access to mesh entities through primitive arrays and opaque entity handles. In addition to global arrays, entities can be accessed individually through iterators in the “Entity” port or in groups of a user-defined size in the “Arr” port. These components also support the the `Tag` interface, which is a generic capability that allows the user to add, set, remove, and change tag information associated with the mesh or individual entities; the `Set` interface, which allows the creation, deletion, and definition of relations among entity sets; and the `Modify` interface, which allows the creation and deletion of mesh entities as well as the modification of vertex coordinate locations.

For the performance studies presented in Section 4.8, we use a simple implementation of the interface that supports two-dimensional simplicial meshes. The mesh software is written in C and uses linked lists to store the element and vertex data. This component has been used primarily as a vehicle for demonstrating the benefits of the component approach to mesh management and for evaluating the performance costs associated therein [95].

4.2 Block-Structured Adaptive Mesh Refinement

Block-structured adaptive mesh refinement (SAMR) is a domain discretization technique that seeks to concentrate resolution where required, while leaving the bulk of the domain sparsely meshed. Regions requiring resolution are identified, collated into rectangular patches (or boxes), and then resolved by a finer grid. Note that the fine grid is *not* embedded in the coarser one; rather, distinct meshes of different resolutions are maintained for the same region in space. Data, in each of these boxes, is often stored as multidimensional Fortran 77 arrays in a blocked format; that is, the same variable (e.g., temperature) for all the grid points are stored contiguously, followed by the next variable. This approach allows operations involving a spatial operator (e.g., interpolations, ghost cell updates across processors) to be written for one variable and reused for others while exploiting cache locality. This approach also allows scientific operations on these boxes to be performed by using legacy codes. The collection of boxes that constitute the discretized domain on a CPU are usually managed by using an object-oriented approach.

SAMR is used by `GraceComponent` [79], a CCA component based on the GrACE library [100], as well as by Chombo [35], a block-structured mesh infrastructure with similar functionality developed by the APDEC [34] group. While each has a very different object-oriented approach (and interface) to managing the collection of boxes, individual boxes are represented in a very similar manner, and the data is stored identically. This fundamental similarity enables a simple, if slightly cumbersome, approach to interoperability.

Briefly, the data pointer of each box is cached in a separate component, along with a small amount of metadata (size of array, position of the box in space, etc.), and keyed to an opaque handle (an integer, in practice). These handles can be exchanged among components, and the entire collection of patches can be recreated by retrieving them from the cache. The interoperability interface is easy to understand and implement; however, the frequent remaking of the box container imposes some overhead, though not excessively so, since array data is not copied. Because the main purpose of this AMR interoperability is to exploit specialized solvers and input/output routines in various packages, metadata overhead is not expected to be significant. Further, this approach is not a preferred means of interoperability on an individual-box basis unless the box is large or the operation very intensive. A prototype implementation of this exchange is being used to exploit Chombo's elliptic solvers in the CFRFS combustion application introduced in Section 2.3. Section 6.3 includes further information about the use of SAMR components in this application.

4.3 Parallel Data Management

The effective management of parallel data is a key facet of many PDE-based simulations. The `GlobalArray` component, based on the Global Array library [94, 98], includes a set of capabilities for managing distributed dense multidimensional arrays that can be used to represent multidimensional meshes. In addition to a rich set of operations on arrays, the user can create ghost cells with a specified width around each of the mesh sections assigned to a processor. Once an update operation is complete, the local data on each processor contains the locally held *visible* data plus data from the neighboring elements of the global array, which has been used to fill in the ghost cells. Two types of update operations are provided to facilitate data transfer from neighboring processors to the ghost regions: a collective update of all ghost cells by assuming periodic, or wraparound, boundary conditions and another nonblocking and noncollective operation for updating ghost cells along the specified dimension and direction with the option to include or skip corner ghost cell updates. The first of these two operations was optimized to avoid redundant communication involving corner points, whereas the second was designed to enable overlapping communication involved in updating ghost cells with computations [99].

Unstructured meshes are typically stored in a compressed sparse matrix form, in which the arrays that represent the data structures are one-dimensional. Computations on such unstructured meshes often lead to irregular data access and communication patterns. The `GlobalArray` component provides a set of operations that can be used to implement and manage distributed sparse arrays (see [24, 30]). Modeled after similar functions in the CMSSL library of the Thinking Machines CM-2/5, these operations have been used to implement the NWPhys/NWGrid [123] adaptive mesh refinement code. Additional `GlobalArray` numerical capabilities have been employed with the optimization solvers discussed in Section 4.6 [13, 65].

4.4 Linear Algebra

High-performance linear algebra operations are key computational kernels in many PDE-based applications. For example, vector and matrix manipulations, along with linear solvers, are needed in each of the motivating applications introduced in Sections 2 and 3.3, as well as in the integration and optimization components discussed in Sections 4.5 and 4.6, respectively.

Linear Algebra Interfaces

Linear algebra has been an area of active interface development in recent years. Abstract interfaces were defined in the process of implementing numerical linear algebra libraries, such as the Hilbert Class Library [51], the Template Numerical Toolkit [106], the Matrix Template Library [85], PLAPACK [7], uBLAS (part of the BOOST collection) [25], BLITZ++ [129], and the Linear System Analyzer [27]. Many of these packages were inspired by or evolved from legacy linear algebra software, such as BLAS and LAPACK. This approach allowed the flexibility of object-oriented design to be combined with the high performance of optimized library codes. In some cases, such as BLITZ++, the goal is to extract high performance even if computationally intensive operations are implemented by using high-level language features; in that case, the library assumes a burden similar to that of a compiler in order to ensure that array operations are performed in a way that exploits temporal and spatial data locality.

Starting in 1997, the Equation Solvers Interface (ESI) working group [31] focused on developing interfaces targeted at the needs of large-scale DOE ASCI program computations, but with the goal of more general use and acceptance. The ESI includes interfaces for linear equation solvers, as well as support for linear algebra entities such as index sets, vectors, and matrices.

More recently, the Terascale Optimal PDE Simulation (TOPS) Center [66], whose mission is to develop a set of compatible toolkits of open-source, optimal complexity solvers for nonlinear partial differential equations, has produced a prototype set of linear algebra interfaces expressed in SIDL [114]. This language-independent specification enables a wide variety of new underlying implementations as well as access to existing libraries. Special care has been taken to separate functionality from the details of accessing the underlying data representation. The result is a hierarchy of interfaces that can be used in a variety of ways depending on the needs of particular applications. See section 3.3 for some simple examples and code using linear algebra components based on the TOPS interfaces.

Linear Algebra Component Implementations

As discussed in detail in [95], an early CCA linear solver port was based on ESI [31] and was implemented by two components based on Trilinos [56] and PETSc [10,11]. The creation of basic linear algebra objects (e.g., vectors and matrices) was implemented as an abstract factory, with specific factory implementations based on Trilinos and PETSc provided as components. The factory and linear solver components were successfully reused in several unrelated component-based applications. Linear algebra ports and components based on TOPS [66] interfaces are currently under development. One of the most significant advancements since the original simple linear solver ports and components were developed is the use of SIDL for interface definition, alleviating implementation language restrictions. By contrast, the ESI-based ports and components used C++, making the incorporation in non-C or C++ applications more difficult. The most recent linear solver component implementations, such as those shown in the examples in Section 3.3, are based on the language-independent TOPS interfaces [114].

4.5 Integration

Ordinary differential equations (ODEs) are solved routinely when modeling with PDEs. Often the solution is needed only locally, for example, when integrating stiff nonlinear chemical reaction rates at a point in space over a short time span. At other times we need a large parallel ODE solution to a method-of-lines problem.

By wrapping the CVODE [33] library, we have created the `CVODEComponent` [79, 115] for the solution of local ODEs in operator-splitting schemes, such as in the combustion modeling application introduced in Section 2.3. Like the library it wraps, `CVODEComponent` can be used to solve ODEs by using variable-order Adams-Bashforth or backward-difference formula techniques. The library provides its own linear solvers and requires the application to provide data using a particular vector representation. The application can provide a sparse, banded, or dense gradient or can request CVODE to construct a finite difference approximation of the gradient if needed.

For parallel ODE solutions we have refactored the LSODE [58, 107] library to allow the application to provide abstract linear solvers and vectors. The parallelism of the ODE is hidden from the integration code by the vector and solver implementations. The resulting `IntegratorLSODE` and related components are described in [5]. These components have been coupled with PETSc-based linear algebra components described in Section 4.4 to solve finite element models [95].

4.6 Optimization

The solution to boundary value problems and other PDEs often can be represented as a function $u \in U$ such that $J(u) = \inf_{v \in U} J(v)$. In this formulation, U is a set of admissible functions, and $J : U \to \Re$ is a functional representing the total energy associated with an element in U. This formulation of a PDE is often preferred for nonlinear PDEs with more than one solution. While each solution satisfies the first-order optimality conditions of the corresponding minimization problem, the solution that minimizes the energy functional is often more stable and of greater interest.

The minimization approach also enables inequality constraints to be incorporated in the model. Obstacle problems, for example, have a free boundary that can be modeled by using variational inequalities. Efficient algorithms with rigorous proof of convergence can be applied to minimization problems with inequality constraints [14, 16]. Even PDEs whose corresponding minimization problem is unconstrained or equality constrained can benefit from optimization solvers [90].

Optimization components [113] based on the TAO library [15] encapsulate the algorithmic details of various solvers. These details include line searches, trust regions, and quadratic approximations. The components interact with the application model, which computes the energy function, derivative information, and constraints [65]. The optimization solvers achieve scalable and efficient parallel performance by leveraging general-purpose linear solvers and specialized preconditioners available in external components, including the data management components discussed in Section 4.3 and the linear algebra components described in Section 4.4 [13, 65, 95].

4.7 Parallel Data Redistribution

As discussed in Section 2, scientific simulations are increasingly composed of multiple distinct physics models that work together to provide a more accurate overall system model or to otherwise enhance fidelity by replacing static boundaries with dynamically computed data values from a live companion simulation. Often each of these models constitutes its own independent code that must be integrated, or *coupled*, with the other models' codes to form a unified simulation. This coupling is usually performed by sharing or exchanging certain common or relevant data fields among the individual models, for example, heat and moisture fluxes in a coupled climate simulation. Because most high-performance scientific simulations require parallel algorithms, the coupling of data fields among these parallel codes raises a number of challenges. Even for the *same* basic data field, each distinct model often applies a unique distributed data decomposition to optimize data access patterns in its localized portion of a parallel algorithm. Further, each model can use a different number of parallel processors, requiring complex mappings between disparate parallel resource topologies (hence the characterization of this mapping as the "MxN" problem – transferring data from "M" parallel processors to another set of "N" processors, where M and N are not in general equal). These mappings require both an understanding of the distributed data decompositions for each distinct model, to construct a "communication schedule" of the elements between the source and destination data fields, as well as special synchronization handling to ensure that data consistency is maintained in any MxN exchanges.

Worse yet, each model may compute using a different time step or may store data elements on a unique mesh using wholly different coordinate systems or axes. This situation necessitates the use of complex spatial and temporal interpolation schemes, and often the preservation of key energy or flux conservation laws. Such complications further exacerbate the already complex infrastructure necessary for coupling disparate data arrays, and require the incorporation of a diverse set of interpolation schemes that are often chosen as a part of the system's scientific requirements, or simply to ensure backwards compatibility with legacy code. Some software packages capable of addressing these issues exist, notably the Mesh-based parallel Code Coupling Interface (MpCCI) [1, 2] and the Model Coupling Toolkit (MCT) [71, 73]. The details of interpolation schemes and their inclusion in MxN infrastructure are beyond the scope of the current work. Yet this important follow-on research will commence upon satisfactory completion of the fundamental generalized MxN data exchange technology. In the meantime, such data interpolation must be handled manually and separately by each distinct code or by an intermediary piece of coupling software.

Parallel Data Redistribution Interfaces

The CCA project is developing generalized interfaces for specifying and controlling MxN parallel data redistribution operations [22]. These interfaces and their accompanying prototype implementations address synchronization and data movement issues in parallel component-based model coupling. Initial efforts have focused on independently defining the local data allocation and decomposition information within a given parallel component and then applying these details to automatically generate efficient mappings, or communication schedules, for executing MxN transfers. These evolving interfaces are sufficiently flexible and high-level to minimize the instrumentation cost for legacy codes, and to enable nearly *transparent* operation by most of the coupling participants.

Parallel Data Redistribution Component Implementations

A variety of general-purpose prototype MxN parallel data redistribution components have been developed by using existing technology. Initial prototypes, which were loosely based on the CUMULVS [49, 69] and PAWS [12, 63] systems, provided a proof of concept to verify the usefulness of the MxN interface specification and to assist in evolving this specification toward a stable and flexible standard. The `CumulvsMxN` component continues to be extended to cover a wider range of data objects, including structured and unstructured dense meshes and particle-based decompositions. The underlying messaging substrates for MxN data transfers are also being generalized to improve their applicability to common scientific codes. Additional MxN component solutions are being developed based on related tools such as Meta-Chaos [42, 108] and ECho [43, 136].

Special-purpose MxN components [72] for use with climate modeling simulations have been built using the Model Coupling Toolkit (MCT) [71, 73] (see Section 6.2). These prototype coupler components provide crucial scientific features beyond fundamental parallel data transfer and redistribution, including spatial interpolation, time averaging and accumulation of data, merging of data from multiple components for use in another component, and global spatial integrals for enforcing conservation laws [22]. Currently MCT is being employed to couple the atmosphere, ocean, sea-ice, and land components in the Community Climate System Model [54] and to implement the coupling interface for the Weather Research and Forecasting Model [132]. Similar coupling capabilities for climate modeling are also being explored as part of the Earth System Modeling Framework (ESMF) [67] effort, with specially tailored climate-specific interfaces and capabilities.

4.8 Performance Overhead Studies

Object-oriented programming in general and components in particular adopt a strict distinction between interfaces and implementations of functionality. The following subsections demonstrate that the interface-implementation separation results in negligible overhead when appropriate levels of abstraction are employed.

CCA Components

To quantify the overhead associated with CCA components, we solved an ODE, $Y_t = F(Y,t)$, where Y is a 4-tuple, implicitly in time using the `CVODEComponent` integration component introduced in Section 4.5. The numerical scheme discretizes and linearizes the problem into a system of the form $Ax = b$, which is solved iteratively. A is derived from the Jacobian of the system, $\partial F/\partial Y$, which is calculated numerically by evaluating F repeatedly. Each F invocation consists of one $\log$ evaluation and 40 exponentials and corresponds to the simplest detailed chemical mechanism described in Section 6.3. A three-component assembly was created (the `Driver`, `Integrator` and `F`), and the `F` evaluations were timed. These were then compared with timings from a non-component version of the code. In order to reduce instrumentation-induced inaccuracies, the problem was repeated multiple times and the total time measured. These steps were taken to ensure that the invocation overhead was exercised repeatedly and the time being measured was significant. The code for F was written in C++ and compiled using `g++ -O2` using egcs version 2.91 compilers on a RedHat 6.2 Intel/Linux platform.

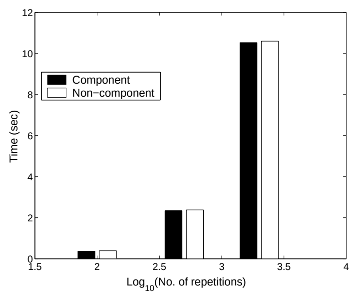

Fig. 13. Timings (in seconds) for component and non-component versions of an ODE application. The differences clearly are insignificant. The x-axis shows the number of times the same problem was solved to increase the execution time between instrumentation invocations.

Figure 13 plots the solution time using the C++ component and the non-component versions. The differences are clearly insignificant and can be attributed to system noise. For this particular case, the function invocation overheads were small compared to the function execution time. Likewise, negligible overhead for both C++ and SIDL variants of optimization components has been shown [13, 95]. The actual overhead of the virtual pointer lookup has been estimated to be of the order of a few hundred nanoseconds (around 150 ns on a 500 MHz Intel processor) [20]. Thus, the overhead introduced by componentization is expected to be insignificant unless the functions are exceptionally lightweight, such as pointwise data accessor methods.

TSTT Mesh Interfaces

We next evaluate the performance ramifications of using a component model for the finer-grained activity of accessing core mesh data structures, where we used the simple mesh management component described in Section 4.1. Since the granularity of access is a major concern, initial experiments focused on the traversal of mesh entities using work set iterators. These iterators allow the user to access mesh entities in blocks of a user-defined size, N. That is, for each call to the iterator, N entity handles are returned in a SIDL array, and it is expected that as N increases, the overhead associated with the function call will be amortized. For comparison purposes,

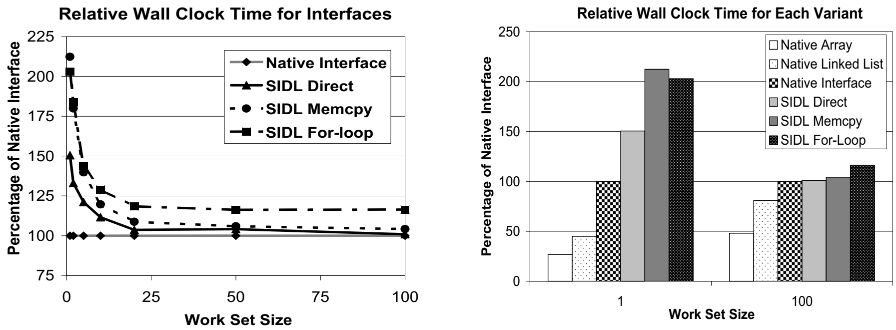

Fig. 14. Average wall clock time for traversing all mesh elements by work set size relative to Native Interface. *(Left)*: A comparison of the four TSTT interface approaches for work set sizes 1 through 100 entities. *(Right)*: A comparison of the six variants for work sets of size 1 and 100 entities only.

experiments were also performed using native data structures to quantify the base costs.

Figure 14 shows the relative costs of obtaining entity handles from the mesh using both native data structures and interfaces. In the experiments, six different mechanisms were used for data access. The native variants consist of timing array (*Native Array*), linked list (*Native Linked List*), and language-specific TSTT interface (*Native Interface*) versions. In order to test the performance of the language interoperability layer created by SIDL, three variations of managing the conversion between native and SIDL arrays were developed. The first two, referred to as *SIDL Direct* and *SIDL Memcpy*, take advantage of the fact that the language interoperability layer and native implementation are both written in C. The former allows the underlying implementation to directly manage the SIDL array contents, while the latter is able to use the `memcpy` routine. The general-purpose variant, called *SIDL For-loop*, individually copies the pointers from the native into the SIDL array. Babel 0.9 was used to generate the interoperability layer for the underlying mesh implementation. The results were obtained on a dedicated Linux workstation with a 1.7 GHz Intel Pentium processor and 1 GB RD RAM using three meshes that ranged in size from 13,000 to 52,000 elements and work set sizes from 1 to 2,000 elements. The codes were compiled *without* optimization, and the timing data was measured in microseconds. Because of the consistency in results across the three different mesh sizes, the average value of all runs on the meshes is reported for each work set size.

The left-hand side of Figure 14 reports the percentage increase for the SIDL-based accesses compared to the baseline native interface access for increasing work set sizes. As expected, the additional function call and array conversion overhead of the SIDL interoperability layer is most noticeable when accessing entities using a work set size of 1 and ranges from 50% more expensive than the native interface for the *SIDL Direct* to 112% more expensive for the *SIDL Memcpy* variant. From work set size 2 on, the *SIDL For-loop* variant is the worst performing. For all cases, the *SIDL Direct* gives the best performance. As the work set size increases to 20 entities and beyond, the SIDL-related overhead decreases to 3.7% more than the native interface in the direct case and approximately 18.4% more in the for-loop case.

To gauge the overhead associated with functional interfaces compared to accessing the data directly using native data structures, the costs of all six access mechanisms are shown on the right-hand side of Figure 14. The results for the interface-based versions are displayed for work set sizes of 1 and 100 entities. As expected, the array traversal is the fastest and is 40% faster than traversing a linked list. Going through the native C interface is about 2.2 and 1.2 times slower than using the linked lists directly for work set sizes 1 and 100, respectively. For work sets of size 1 and 100, the *SIDL Direct* method gives the best SIDL performance and is 3.3 and 1.2 times slower, respectively, than using linked lists. For work sets of size 1, the memcpy SIDL variant is 4.7 times slower (versus 4.5 times slower for the for-loop version). This position is reversed for work sets of size 100, where the for-loop version is 44% slower (versus 29% for the memcpy version).

These experiments demonstrate that the granularity of access is critical in determining the performance penalty involved in restructuring an application to use an interface-based implementation. However, the granularity does not need to be very large. In fact, our experiments show that work sets of size 20 were sufficient to amortize the function call overhead. We also found that the additional overhead associated with transitioning from a native to language interoperable version of the interfaces can be negligible for suitable work set sizes.

5 Componentization Strategies

Next we examine strategies for developing software components for parallel PDE-based applications, including projects that incorporate legacy code as well as completely new undertakings. These general considerations have been employed when developing the reusable components discussed in Section 4 as well as throughout the application case studies presented in Section 6. Useful component designs may be coarse-grained (handling a large subset of the overall simulation task per component) or fine-grained. Similarly, an interface (port) between components may be a simple interface with just a few functions having a few simple arguments or a complex interface having all the functions of a library such as MPI.

The first step in defining components and ports for PDE-based simulations is considering in detail the granularity and application decomposition of the desired software. The second step is evaluating the impact on implementers and users of the chosen component and interface designs. This step may involve implementation and testing. The third step is iterating the design and implementation steps until a sufficient subset of the implementers and users is content with the resulting components and interfaces. The finest details of CCA component software design [5] and construction [21] are beyond our scope.

5.1 Granularity

How much functionality is inside each component? What information appears in public interfaces? In what format does the information appear? The answers to these questions determine the granularity of any component. We may want fine or coarse granularity or a combination.

Very coarse-grained designs. At the coarsest granularity, an entire PDE simulation running on a parallel machine can be treated as a single component. In most cases of this scenario, the component simply uses input files and provides output files. This approach permits the integration of separate programs by applying data file transformations. The overhead of transferring data using intermediate files may be reduced by instead transporting data directly from one application to another in any of several ways, but the essential aspect of copying data from one simulation stage to the next remains (see Section 4.7). The cost of moving large data sets at each iteration of an algorithm to and from file systems in large parallel machines can be prohibitive. Nonetheless, the CCA specification allows components to interact by file exchange. Software integration through files (see, e.g., Section 6.1) can be a useful starting point for the evolutionary development of a coupled simulation. Many tools have been built on this style of componentization, but scaling up to very large, multidomain, multiphysics problems is often inefficient or even impossible.

Finer-grained designs. For the remainder of this section we are concerned with building an application from component libraries that will result in executing a single job. Nearly every PDE-based application has one or more custom drivers that manage a suite of more general libraries handling different areas of concern, such as mesh definition, simulation data, numerical algorithms, and auxiliary services, such as file input and output, message passing, performance profiling, and visualization. Building component software for PDEs means teasing apart these many areas of concern into separate implementations and defining suitable functional interfaces for exchanging data among these implementations. Once clearly separated, the substitution and testing of an alternate implementation in any individual area is much easier.

5.2 Application Decomposition and Interface Design

Making components for a new or existing PDE-based simulation will be straightforward or difficult depending on how well the code is already decomposed into public interfaces and private implementations. We have found that the answers to the following technical questions about a PDE software system can aid in creating a good first approximation to an equivalent component design. Like all good software design, an iterative implementation and evaluation process is required to arrive at a final design.

- What are the equations to be solved, the space and time domains in which these equations apply, and the kinds of boundary and initial conditions that apply at the edges of these domains?

- What are appropriate meshing and spatial discretization techniques and an efficient machine representation of the resulting mesh?
- How is data stored for the simulation variables, and how are relationships of the stored data to various mesh entities (points, edges, faces, cells) represented?
- What are appropriate algorithms for solving the equations?
- What are appropriate implementations of the algorithms, given the algorithms and data structures selected? Can these implementations be factored into multiple independent levels, such as vector and matrix operations, linear solvers, nonlinear solvers, and time marching solvers?
- How can we identify, analyze, and store interesting data among the computed results?

Usually many mathematical and software design solutions exist for each of these questions, and of course they are interconnected. Our fundamental goal in component design for an application involving large-scale or multidomain PDEs is to separate these areas of concern into various software modules without reducing overall application performance. Many packages used for solving PDEs are already reasonably well decomposed into subsystems with clear (though often rather large) interfaces that do not impede performance; extending these packages to support component programming is usually simple unless they rely on global variables being set by the end user.

The performance requirement directly impacts the interfaces *between* components. Arrays and other large data structures that are operated on by many components must be exchanged by passing pointers or other lightweight handles. This requirement in turn necessitates specific public interface commitments to array and structure layout, which depend on the type of PDE and the numerical algorithms being used. CCA componentization allows similar packages to work together, but it does not provide a magic bullet solution to integrating packages based on fundamentally different assumptions and requirements. For example, a mesh and data component representing SAMR data as three-dimensional dense arrays [77, 79] is simply inappropriate for use in adaptive finite-element algorithms that call for data trees. When constructing new components, it is useful for interoperability to design them to be as general as reasonably possible with respect to details of the data structures they can accept. For example, we suggest dealing with arrays specified by strides through memory for each dimension rather than restricting a code to either row-major or column-major layout.

Some preprocessing or postprocessing components may demand flexibility in accessing many different kinds of mesh or data subsets. In this situation public interface definitions that require a function call to access data for each individual node or other mesh entity may be useful. However, as seen in testing the TSTT mesh interfaces (Section 4.8), single entity access function overhead slows inner loops, so that care should be taken when deciding to use such interfaces.

Success is often in the eye of the beholder when choosing a software decomposition and when naming functions within individual interfaces. Compact, highly reusable objects, such as vectors and other simple objects in linear algebra, may appear at first to be tempting to convert into components or ports. Instead, in many CCA applications, experience has shown that these objects are best handled as function arguments. Many high-performance, component-based implementations for PDEs have been reported. Most feature a combination of a mesh definition component, a data management component, and various numerical algorithm components [4, 21, 72, 78, 95, 138].

5.3 Evaluating a Component System Design

Each reusable component provides a set of public interfaces and may also use interfaces implemented by other components. Armed with our technical answers about the PDE software system that we will wrap, refactor, or create from scratch, we must ask questions about how we expect the components to be used and implemented. Is some aspect of the design too complex to be useful to the target users? Have we introduced an interface in such a way that we lose required efficiency or capability?

Component granularity checks. An application is formed by connecting a set of components and their ports together, and the more components involved in an application, the more complex it is to define. Too many components *may* be an indication of an overly fine decomposition. A good test is to consider the replacement of each component individually with an alternative implementation. If this would necessitate changing multiple components at once, then that group may be a candidate for merging into a single component. As a rule of thumb, we have found that if a *simple* test application, such as those discussed in Section 3.3, requires more than seven components, then the decomposition may be worth revisiting.

Similarly, a component providing or using too many ports may be an indication that it contains too much functionality to be manageable and should be decomposed further. Multiple ports may offer alternative interfaces to the same functionality, but many unrelated ports may signal a candidate for further study. A detailed case study of component designs for ODE integrations is given in [5].

User experience. Empirical evidence determined by CFRFS combustion researchers, whose work is introduced in Section 2.3, indicates that the final granularity of a component-based application code is arrived at iteratively. One starts with a coarse-grained decomposition, which in their case was dictated by the nature of their time-integration scheme, and moves to progressively more refined and fine-grained designs. For the CFRFS researchers, the more refined decompositions were guided by physics as represented as mathematical operators and terms in the PDE system being solved. The finest granularity was achieved at the level of physical/chemical models. For example, in their flame simulation software, the transport system formed an explicit-integration system. In the second level of refinement, transport was separated into convection and diffusion, which occurs as separate terms in their governing equations. In the final decomposition, diffusion was separated into a mathematical component that implemented the discretized diffusion operator, while the functionality of calculating diffusion coefficients (required to calculate the diffusion term and used in the discretized diffusion operator) was separated as a specialized component [79]. Such a decomposition enables the testing of various diffusion coefficient models and discretizations by simply replacing the relevant component. Since these components implement the same ports, this activity is literally plug-and-play.

Port complexity checks. Many ports for PDE computations are as simple as a function triplet (setup, compute, finish) with a few arguments to each function (see, for example, the climate component discussion in Section 6.2). Port interface design complexity can be measured in terms of the number of functions per port and the number of arguments per function. If both these numbers are very high, the port may be difficult to use and need further decomposition. Should some of the functions instead be placed in a component configuration port that is separate from the main computation function port? Is a subset of the functions in the port unique to a specific implementation, making the port unlikely to be useful in any other component? Are there several subsets of functions in the port such that using one subset implies ignoring other subsets? Conversely, there may be so few functions in a port that it is always connected in parallel with another port. In this case the two ports may be combined.

5.4 Adjusting Complexity and Granularity

Of course a degree of complexity is often unavoidable in the assembly of modern applications. The CCA specification provides for *containers*, which can encapsulate assemblies of components to further manage complexity [19]. This capability allows a component set to appear as a single component in an application, so that the granularity of components and complexity of interfaces can be revised for new audiences without major re-implementation. The container may expose a simplified port with fewer or simpler functions (and probably more built-in assumptions) than a similar complex port appearing on one of its internally managed components. Some or all of the ports not connected internally may be forwarded directly to the exterior of the container for use at the application level.

6 Case Studies: Tying Everything Together

The fundamental contribution of component technology to the four parallel PDE-based applications introduced in Section 2 and discussed in detail in this section is the enforcement of modularity. This enforcement is *not* the consequence of programming discipline; rather, it is a fundamental property of the component paradigm itself. The advantages observed are those naturally flowing from modularization:

1. *Maintainability*: Components divide complexity into manageable chunks, so that errors and substandard implementations are localized and easily identifiable, and consequently may be quickly repaired. Further, the consequences of careless design of one component often stop at its boundary.
2. *Extensibility*: Modularization limits the amount of detail that one has to learn before beginning to contribute to a component-based application. This simplifies and accelerates the process of using and contributing to an external piece of software and thus makes it accessible to a wider community.

3. *Consistency*: Even though the component designs for the various projects have been agreed to rather informally within small communities, using the component-based architecture ensures through compile-time checking that object-oriented, public interfaces (ports) are used *consistently* everywhere. This approach eliminates errors often associated with older styles of interface definition such as header files with global variables and C macros that may depend on compiler flags or potentially conflicting Fortran common block declarations in multiple source files. For example, in the CFRFS project introduced in Section 2.3, interfaces and the overall design could be (and was) changed often and without much formal review. However, each port change *had to be* propagated through all the components dependent on that port interface in order for the software to compile and link correctly. This requirement enforced uniformity and consistency in the design. As the CFRFS toolkit grew, major interface changes became more time-consuming, compelling the designers to design with care and completeness in the first place, a good software engineering practice under any conditions.

We now discuss how component technology has been applied in these four scientific applications, each of which faces different challenges and is at a different stage of incorporating component concepts. We begin in Section 6.1 by discussing the accelerator project, introduced in Section 2.1, which is currently at an early phase of exploring a component philosophy for mesh infrastructure to facilitate experimentation with different meshing technologies, as introduced in Section 4.1. Section 6.2 explains how climate researchers have decomposed their models using the general principles introduced in Section 5 to develop next-generation prototype applications that handle model coupling issues, which were introduced in Sections 2.2 and 4.7. Section 6.3 highlights how the plug-and-play nature of CCA components enables combustion scientists to easily explore different choices in algorithms and data structures and thereby achieve their scientific goals, introduced in Section 2.3. The final application, discussed in Section 6.4, explains how CCA components help to harness the complexity of interdisciplinary simulations involving accidental fires and explosions, as introduced in Section 2.4. Here components allow diverse researchers to work together without being in lock step, so that a large, multiphysics application can achieve efficient and scalable performance on a wide range of parallel architectures.

6.1 Accelerator Modeling

As introduced in Section 2.1, ongoing collaborations among scientists in the TSTT Center and SLAC have resulted in a number of improvements to the mesh generation tools and software infrastructure used for accelerator modeling. Although the TSTT mesh interfaces are not yet mature enough for direct use in an application code, a component philosophy is being employed to insert TSTT adaptive mesh capabilities into the finite element-based frequency domain code, Omega3P.

Initially, the goal was to demonstrate the benefits of adaptive mesh refinement without changing a line of code in the core of Omega3P. This goal was accomplished by cleanly dividing the responsibilities of the different pieces of software and iteratively processing the mesh until convergence was achieved. In particular, TSTT tools developed at RPI handled error estimation and adaptive refinement of the mesh, while Omega3P computed the solution fields. Information was exchanged between Omega3P and the TSTT meshing tools by using a file-based mechanism in which the current mesh and solution fields were written to a file that was then read by the RPI tools.

Although the performance of a file-based mechanism for information transfer between adaptive refinement steps is clearly not ideal, it proved to be an excellent starting point because it allowed a very quick demonstration of the potential benefits of adaptive mesh refinement for the Omega3P code. As mentioned in Section 2.1, a high degree of accuracy is required in the frequency domain results. For one commonly used test geometry, the Trispal geometry, the results from the adaptive refinement loop were more accurate than those from prior simulations and provided the best match to experimental results at a fraction of the computational cost [47]. Furthermore, this work has showcased the benefits of modularizing various aspects of the simulation process by allowing SLAC researchers to quickly use AMR technologies without requiring a wholesale change to their code. Based on the success of this demonstration, work is now proceeding to insert the adaptive refinement loop directly into Omega3P using the TSTT interface philosophy and the underlying implementations at RPI.

6.2 Climate Modeling

In Section 2.2 we described how the climate system's complexity leads to software complexity in climate system models. Here we discuss in greater detail the practices of the climate/weather/ocean (CWO) community; for brevity, our scope is restricted to atmospheric global climate models. We discuss the refactoring of CWO software to make it more component friendly and to alleviate complexity. We describe the CWO community's effort to create its own component specification (ESMF). We also present a prototype component-based atmospheric advection model, which uses both the ESMF and CCA paradigms. See [72] for further information on these topics.

Model Decomposition

As mentioned in Section 2.2, the fundamental equations for atmosphere and ocean dynamics are called the primitive equations. Their solvers are normally structured in two parts. The first part, which solves the primitive equations, is called the *dynamics*, *dynamical core*, or *dycore*. The second part, which models source and sink terms resulting from length scales shorter than those used in the dynamical core's discretization, is called the *physics*. Examples of parameterized physical processes in the atmosphere include convection, cloud formation, and radiative transfer. In principle, one could use the same solver infrastructure for both atmosphere and ocean dynamics, but this approach is rarely used because of differences in model details.

Climate models are a natural application for component technology. Figure 15 illustrates one of the ways for the component decomposition at several levels. The

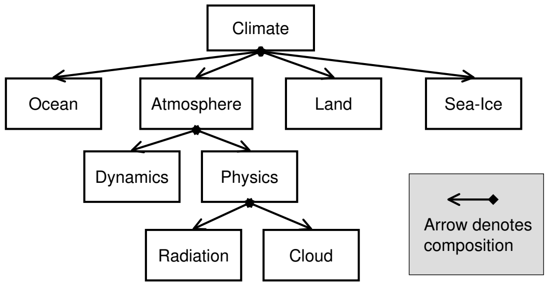

Fig. 15. Diagram of climate models decomposed in terms of components.

highest level integrates the major subsystems of the earth's climate (ocean, atmosphere, sea-ice, and land-surface), which are each a component. Within the atmosphere, we see a component decomposition of the major parts of the model—the dynamics and the physics. Within the physics parameterization package, each subgridscale process can also be packaged as a component.

The plug-and-play capabilities of component-based design, as introduced in Section 3, can aid researchers in exploring trade-offs among various choices in meshing, discretization, and numerical algorithms. As an example we consider atmospheric global climate models, in which various approaches, each with different advantages and disadvantages, can be used for solving the primitive equations. The main solvers are finite-difference [41] and spectral methods [50]; semi-Lagrangian and finite element techniques are also sometimes used. Various solvers have been developed for each approach, including the Aries dycore [118], the GFDL and NCAR spectral dycores [50,126], and the Lin-Rood finite-volume dycore [83]. An additional challenge is handling the physical mesh definition; choices for the horizontal direction include logically Cartesian latitude-longitude, geodesic [36], and cubed-sphere [86]. Various choices for the vertical coordinate include pressure, sigma-pressure, isentropic, or a combination of these.

A primary challenge in developing atmospheric models is achieving scalable performance that is portable across a range of parallel architectures. For example, parallel domain decompositions of atmospheric dycores are usually one- or two-dimensional in the horizontal direction. Most codes use MPI or an MPI/OpenMP hybrid scheme for parallelization because it provides solid performance and portability. Component-based design helps to separate issues of parallelism from portions of code that handle physics and mathematics, thereby facilitating experimentation with different parallel computing strategies. Another challenge is language interoperability. The modernization path for most climate models has been a migration from Fortran 77 to Fortran 90, combined with some refactoring to increase modularity. Few of these models are implemented in C or C++ [122]. As discussed in Section 3, the programming language gap between applications and numerical libraries is an issue that component technology can help to bridge.

In response to a CWO component initiative, scientists have been refactoring their application codes. Initially, this activity was undertaken simply to provide better modularity and sharing of software among teams. A good example is the refactoring of the Community Atmosphere Model (CAM) to split the previously entangled physics and dynamics portions of code. This entanglement made the change of one dycore for another an arduous process. Now that the physics and dynamics have been split, CAM has three dycores: the spectral dycore with semi-Lagrangian moisture transport, the Lin-Rood finite-volume dycore, and the Williamson-Rasch semi-Lagrangian scheme. This refactoring will ease the integration and testing of the newly developed NCAR spectral element dycore. An effort is also under way to repackage these dycores as ESMF components (see below), which will be the first introduction of component technology to CAM.

Model Coupling

The primitive equations are a boundary-value problem. For the atmosphere, the lateral boundary values are periodic, the top of the atmosphere's boundary condition is specified, and the boundary conditions at the Earth's surface are provided by the ocean, sea-ice, and land-surface components. This mutual interaction between multiple subsystems requires MxN parallel data transfers, such as those described in Section 4.7. This need for boundary data also poses the problem of how to schedule and execute the system's atmosphere, ocean, sea-ice, and land-surface components to maximizes throughput. There are two basic scheduling strategies. The first is a sequential event-loop strategy, in which the components run in turn on the same pool of processors (e.g., the Parallel Climate Model (PCM) [23]). The second strategy is concurrent component execution, in which each model executes independently on its own pool of processors (e.g., CCSM). Componentization of the land, atmosphere, ocean, and sea-ice models will increase overall flexibility in scheduling the execution of a climate model's constituents and thereby facilitate aggressive experimentation in creating previously unimplemented climate system models.

ESMF and the CCA

The great potential that components offer for enabling new science has inspired the CWO community embrace component technology. Of particular note is the NASA-funded interagency project to develop the Earth System Modeling Framework [57, 67]. The ESMF comprises a *superstructure* and an *infrastructure*. The superstructure provides the component specification and the overall component interfaces used in coupling. The infrastructure includes commonly needed low-level utilities for error handling, input/output, timing, and overall time management. The infrastructure also provides a common data model for coordinate grids, physical meshes, and layout of field data, as well as services for halo updates, parallel data transfer, and intergrid interpolation, much like the facilities described in Section 4.7. One distinguishing feature of ESMF components is that they have three methods: `Initialize`, `Run`, and `Finalize`. The ESMF supplies its coupling data in the form of the `ESMF_State` datatype.

ESMF developers have collaborated with the CCA to ensure framework interoperability, so that ESMF components may run in a CCA-compliant framework and vice versa. This effort will provide scientists application-specific ESMF services in composing climate components, while also enabling the use of CCA numerical components, such as those described in Section 4.

A Prototype Component-Based Climate Application

A prototype coupled atmosphere-ocean application, which employs both the CCA and ESMF component paradigms, has been developed as proof-of-concept application for CWO codes [139, 140]. The application combines the CCA component registration infrastructure and *uses-provides* interaction model introduced in Section 3 with the ESMF's component method specification (i.e., `Initialize`, `Run`, `Finalize`) and data model. (i.e., `ESMF_State`). This application includes a component common to the atmosphere and ocean dycores, namely, two-dimensional advection of a quantity Ψ by the horizontal velocity field (u, v):

$$\frac{\partial \Psi}{\partial t} + u\frac{\partial \Psi}{\partial x} + v\frac{\partial \Psi}{\partial y} = S,$$

where $\Psi(x, y, t)$ is the advected quantity, and $S(x, y, t)$ is the sum of all sources and sinks. The x-y spatial grid is rectangular, and the discretization method is a finite-difference scheme. Here we consider three finite difference variants, which are each forward in time and either forward-, central-, or backward-difference in space.

Following the CCA component specification, we created an advection component with a solver port definition for a finite-difference scheme. The advection equation can be solved by using forward-, central-, or backward-differencing in space. We employ a *proxy* design pattern [46] to allow the atmospheric model component the choice of one of these default solvers or a user-designated scheme. We also use CCA technology to enable the user to specify run-time parameters such as advection speed. The ability to easily swap in different implementations in this component-based advection application has proven useful in exploring differences in the accuracy and computational complexity of the various numerical methods.

6.3 Combustion

The objective of the CFRFS [93] project introduced in Section 2.3 is the creation of a component-based toolkit for simulating laboratory-sized (0.1^3 m) flames with detailed chemistry. Such flames contain tens of species, hundreds of reactions, spatial structures 10^{-4} meters in size, and timescales ranging from 10^{-9} seconds (chemical processes) to 10^{-1} seconds (convective processes). The low Mach Navier-Stokes equation [92, 133], and the equations for species' evolution comprise a set of coupled PDEs of the form

$$\frac{\partial \mathbf{\Phi}}{\partial t} = \mathbf{F}(\mathbf{\Phi}, \nabla\mathbf{\Phi}, \nabla^{\mathbf{2}}\mathbf{\Phi}, ...) + \mathbf{G}(\mathbf{\Phi}),$$

where $\mathbf{\Phi}$ consists of flow quantities such as density and temperature. The equation is discretized on rectangular meshes and solved in rectangular domains. For these systems $\mathbf{G}$ involves the variables only at a given mesh point, while $\mathbf{F}$, which involves spatial derivatives (computed by using finite-difference or finite-volume schemes), depends on the mesh point and its close neighbors. $\mathbf{G}$ is stiff, so that the ratio of the largest and the smallest eigenvalues of $\partial\mathbf{G}/\partial\mathbf{\Phi}$ is large, while $\mathbf{F}$ is non-stiff. Operator splitting [116, 117] is employed to evolve the stiff ($\mathbf{G}$) and nonstiff ($\mathbf{F}$) terms in a decoupled manner by following a $\mathbf{GFG}$ sequence, thus letting the stiff operator be the last in the time step, in order to achieve higher accuracy in the data reported at the end of the time step. A backward-difference formulation [33] and a Runge-Kutta-Chebyshev integrator [9] are used for the stiff and nonstiff problem, respectively. The solution vector $\mathbf{\Phi}$ exhibits steep spatial variations in scattered, time-evolving regions of the domain. Block-structured adaptive mesh refinement (SAMR) [17] and time refinement [18] are used to track and resolve these regions.

The CFRFS team used CCA-compliant component technology to explore the use of high-order spatial discretizations in a SAMR setting [76, 78, 110] for the first time and to perform scalability studies of reacting flow problems on SAMR meshes.

High-Order Spatial Discretizations and Block SAMR

PDEs can be discretized by a variety of methods [103]. Finite differences and volumes are popular for solving fluid flows. Typically, second-order spatial discretizations are used, although high-order spatial discretizations on *single-level* structured or unstructured meshes are becoming common [32, 62, 81, 82, 130, 131]. The CFRFS team explored the use of high-order (> 2) schemes in *multilevel* block-structured adaptive meshes [76, 78]. Multilevel, block-structured meshes are a conceptually elegant way of achieving resolution in simple domains. One starts with a coarse, structured, logically rectangular mesh. Regions requiring resolution are identified, collated into patches, and overlaid with a rectangular mesh of higher density. This high-density patch is not embedded; rather, it is preserved separately as a fine patch. This process is carried out recursively, leading to a hierarchy of patches, that is, a multilevel grid hierarchy [18]. In this way a given point in space is resolved at different resolutions simultaneously, by different levels of the grid hierarchy.

Deep grid hierarchies pose significant load-balancing problems. High-order spatial discretizations present a simple solution because they may provide an acceptable degree of accuracy on relatively coarse meshes (i.e., with relatively shallow grid hierarchies). Incorporating high-order schemes in a SAMR setting is nontrivial, however, as the software infrastructure associated with parallel SAMR codes is very complex. Indeed, the mathematical complexities of high-order schemes have restricted their use to relatively simple problems. The component design established a clear distinction between the various domains of expertise and a means of incorporating the contributions of diverse contributors without imposing a programming and data-structural straitjacket. Most contributions were written by experts in Fortran 77 and then componentized.

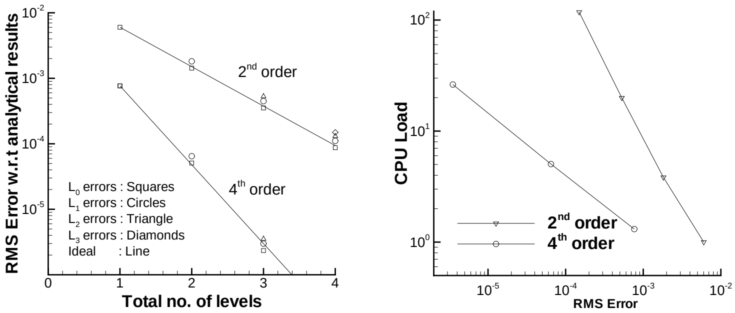

Fig. 16. *(Left)*: The root mean squared (RMS) error on the individual levels, as the simulation is run on a 1-, 2-, 3- and 4-level grid hierarchy. *(Right)*: The computational load versus RMS error for the second- and fourth-order approaches. Results have been normalized by the computational load of a second-order, 1-level grid hierarchy run.

These components were used to simulate PDEs on multilevel meshes, with factor-of-two refinements between levels. The left-hand side of Figure 16 shows the recovery of the theoretical convergence rate as the effective resolution was increased by increasing the number of levels. The base grid has 100 cells in the [0,1] domain. Both second- and fourth-order discretizations were used. High-order schemes were found to be more economical than second-order schemes because they required sparser meshes to achieve a given level of accuracy. Further, higher-order schemes become progressively more economical vis-a-vis second-order approaches as the error tolerances become stringent. This behavior is evident in the right-hand side of Figure 16, which plots the computational loads (in terms of floating point operations count) normalized by the one-level grid hierarchy load (1,208,728,001 operations).

Strong-Scalability Analysis of a SAMR Reacting Flow Code

SAMR scalability studies are rare [134, 135] and usually specific to the applications being tested, that is, specific to the implemented algorithm, and hence are difficult to analyze and interpret. To explore the behavior of the parallel CCA-based block-SAMR toolkit and to identify the scalability bottlenecks, the CFRFS team performed a *strong scaling* study (i.e., the global problem size was kept constant while the number of processors increased linearly) for a two-dimensional reaction-diffusion problem with detailed hydrogen-air chemistry using three levels of refinement with a refinement factor of two [80]. The initial condition was a random distribution of temperature kernels in a stoichiometric hydrogen-air mixture. For this experiment, the parallel virtual machine expanded by a factor of two, starting with 7 processors and reaching 112 processors. Time and messaging volumes were measured by connecting the TAU [127] performance analysis component to the CFRFS component code assembly.

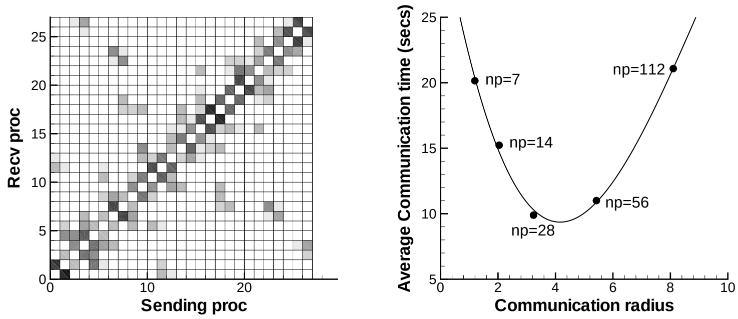

Fig. 17. *(Left)*: Communication patterns for 28 processors at timestep 40. *(Right)*: Communication costs as a function of the communication radius at timestep 40.

Results indicated that the overall scalability of the adaptive algorithm is highly dependent on the scalability of the intermediate time steps [80]. There were "scalable" and "nonscalable" time steps, depending on the quality of the domain decomposition. The nonscalable time steps were a consequence of synchronization times, where many processors idled because of severely uneven computational load partitioning. The scalable time steps showed good load-balance, but their communication times increased as the number of processors increased. The left-hand side of Figure 17 shows the communication map for a 28-processor run. While the bulk of the communication is with the nearest neighbors, there are a significant number of outliers. The remoteness of these outliers was characterized by an average communication radius r. The right-hand side of Figure 17 shows that the average communication time per processor increases with r (after $r \sim 4$), a counterintuitive result, as increasing r indicates more processors and smaller per-processor problem sizes. The explanation lies in the network topology. The scaling study was performed on a cluster with Myrinet, which has a Clos-network topology. Eight nodes are connected to a switch; a cascade of 16-port switches ensures full connectivity, though at increasing levels of indirection. As $r \to 8$, increasing fractions of the total communication occur over the cascade, as opposed to in-switch communication. This situation results in message contentions and collisions and hence slower transfer speeds and larger communication costs.

6.4 Accidental Fires and Explosions

The simulation environment for the Center for the Simulation of Accidental Fires and Explosions (C-SAFE) [55], introduced in Section 2.4, is the Uintah Computational Framework (UCF) [39], which is a set of software components and libraries that facilitate the parallel simulation of PDEs on structured adaptive mesh refinement (SAMR) grids. The UCF is implemented in the context of the CCA-based SCIRun2 framework [138], which supports a wide range of computational and visualization applications. One of the challenges of creating component-based PDE software is achieving scalability, which is a global application property, through components that, by definition, make local decisions.

Managing Parallelism via Taskgraphs

To address this challenge of managing parallelism in multidisciplinary applications, the UCF employs a nontraditional approach. Instead of using explicit MPI calls throughout each component of the program, applications are cast in terms of a *task-graph*, which describes the data dependencies among various steps of the problem.

Computations are expressed as directed acyclic graphs of *tasks*, each of which produces some output and consumes some input, which is in turn the output of some previous task. These inputs and outputs are specified for each patch in a structured AMR grid. Associated with each task is a method that performs the actual computation. This representation has many advantages, including efficient fine-grained coupling of multiphysics components, flexible load balancing mechanisms, and a separation of application and parallelism concerns. Moreover, UCF data structures are compatible with Fortran arrays, so that application writers can use Fortran subroutines to provide numerical kernels on each patch.

Each execution of a taskgraph integrates a single timestep, or a single nonlinear iteration, or some other coarse algorithmic step. Tasks communicate with each other through an entity called the *DataWarehouse*. The DataWarehouse is accessed through a simple name-based dictionary mechanism, and it provides each task with the illusion that all memory is global. If the tasks correctly describe their data dependencies, then the data stored in the DataWarehouse will match the data (variable and region of space) needed by the task. In other words, the DataWarehouse is an abstraction of a global single-assignment memory, with automatic data lifetime management and storage reclamation. Values stored in the DataWarehouse are typically array-structured. Communication is scheduled by a local algorithm that approximates the true globally optimal communication schedule. Because of the flexibility of single-assignment semantics, the UCF is free to execute tasks close to data or move data to minimize future communication.

The UCF storage abstraction is sufficiently high level that it can be efficiently mapped onto both message-passing and shared-memory communication mechanisms. Threads sharing a memory can access their input data directly; single-assignment dataflow semantics eliminate the need for any locking of values. Threads running in disjoint address spaces communicate by a message-passing protocol, and the UCF is free to optimize such communication by message aggregation. Tasks need not be aware of the transports used to deliver their inputs, and thus UCF has complete flexibility in control and data placement to optimize communication both between address spaces or within a single shared-memory symmetric multiprocessing node. Latency in requesting data from the DataWarehouse is not an issue; the correct data is deposited into the DataWarehouse before each task is executed.

Consider the taskgraph in Figure 18. Ovals represent tasks, each of which is a simple array-based subroutine. Edges represent named values stored by the UCF. Solid edges have values defined at each material point, and dashed edges have values

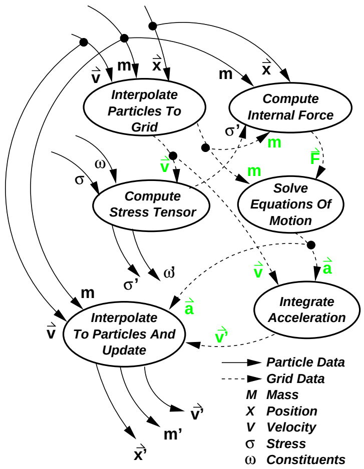

Fig. 18. An example UCF taskgraph, depicting a portion of the material point method (MPM) algorithm used to simulate solid materials in C-SAFE scenarios.

defined at each grid vertex. Variables denoted with a prime (’) have been updated during the time step. The figure shows a portion of the Uintah material point method (MPM) [119] taskgraph concerned with advancing Newtonian material point motion on one patch for a single time step.

The idea of the dataflow graph as an organizing structure for execution is well known. The SMARTS [128] dataflow engine that underlies the POOMA [111] toolkit shares goals and philosophy with the UCF. Sisal compilers [45] used dataflow concepts at a much finer granularity to structure code generation and execution. Dataflow is a simple, natural, and efficient way of exposing parallelism and managing computation and is an intuitive way of reasoning about parallelism. What distinguishes implementations of dataflow ideas is that each caters to a particular higher-level presentation. SMARTS is tailored to POOMA’s C++ implementation and stylistic template-based presentation. The UCF supports a presentation catering to C++ and Fortran-based mixed particle/grid algorithms on structured adaptive meshes, and the primary algorithms of importance to C-SAFE are the MPM and Eulerian computational fluid dynamics algorithms.

This dataflow-based representation of parallel computation fits well with the structured AMR grids and with the nature of the computations that C-SAFE performs. In particular, we used this approach in order to accommodate multiphysics integration, load-balancing, and mixed thread/MPI programming. A more detailed discussion of these advantages (and disadvantages) can be found in [101].

The most important advantage for a large interdisciplinary project such as C-SAFE is that the taskgraph facilitates the separate development of simulation components and allows pieces of the simulation to evolve independently. Because C-SAFE is a research project, we need to accommodate the fact that most of the software is still under development. The component-based architecture allows pieces of the system to be implemented in a basic form at first and then to evolve as the technologies mature. Most importantly, the UCF allows the aspects of parallelism (schedulers, load-balancers, parallel input/output, and so forth) to evolve independently of the simulation components. This approach allows the computer science effort to focus on these problems without waiting for the completion of the scientific applications or vice-versa.

Components Involved

Figure 19 shows the main components involved in a typical C-SAFE simulation. The simulation controller component, which is in charge of the simulation, manages restart files if necessary and controls the integration through time. First, it reads the specification of the problem from an XML input file. After setting up the initial grid, it passes the description to the simulation component, which can implement various algorithms, including one of two different CFD algorithms, the MPM algorithm, or a coupled MPM-CFD algorithm. The simulation component defines a set of tasks for the scheduler. In addition, a data-archiver component describes a set of output tasks to the scheduler. These tasks save a specified set of variables to disk. Once all tasks are known to the scheduler, the load-balancer component uses the machine configuration to assign tasks to processing resources. The scheduler uses MPI for communication and then executes callbacks to the simulation or data-archiver components to perform the actual work. This process continues until the taskgraph has been fully executed. The execution process is then repeated to integrate further time steps.

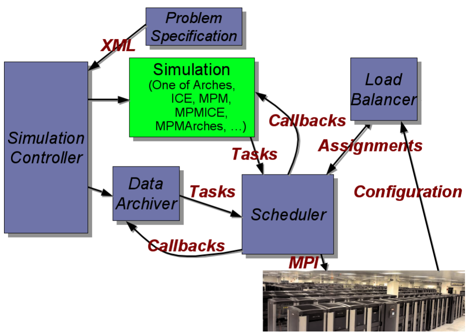

Fig. 19. UCF simulation components. The simulation describes tasks to a scheduling component, which are assigned to processing resources by a load-balancer component. Callbacks are made into the simulation component to perform the computation. Checkpointing and data input/output are performed automatically by the data-archiver component.

Each of these components runs concurrently on each processor. The components communicate with their counterparts on other processors using MPI. However, the scheduler is typically the only component that needs to communicate with other processors. Figure 20 demonstrates that the resulting system scales well on various parallel architectures. Delegating responsibility for parallelism to the scheduler component allows complex multiphysics applications to utilize processor resources efficiently and reduces the programming burden for applications that require complex communication patterns to achieve good scalability.

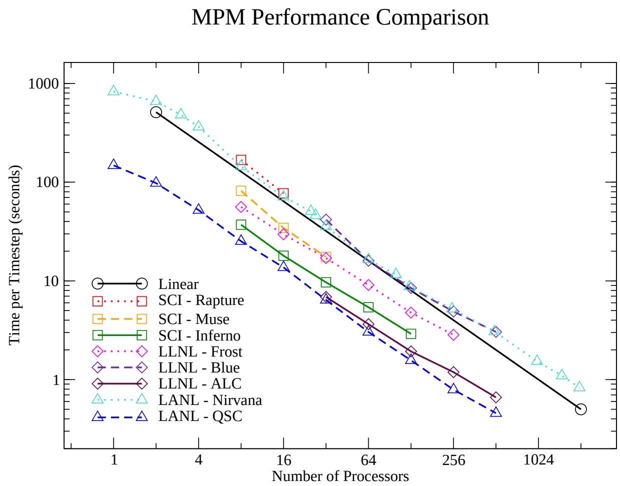

Fig. 20. Performance of the UCF MPM simulation on various architectures during the development of the simulation. Rapture: 32-processor 250 MHz Origin 2000, Muse: 64-processor 600 MHz Origin 3000, Inferno: 256-processor 2.4 GHz Pentium 4 Linux cluster, Frost: 1024-processor IBM SP Power 3, Blue: 3392-processor IBM SP PowerPC 604e, ALC: 1024-processor 2.4 GHz Pentium 4 Linux cluster, Nirvana: 2048-processor 250 MHz Origin 2000 at LANL, QSC: 256-processor 1.25 GHz Alpha. Data courtesy of Randy Jones, Jim Guilkey, and Todd Harman of the University of Utah.

7 Conclusions and Future Work

All component-based approaches to software seek to divide the inherent complexity of large-scale applications into sizes that human beings can deal with individually, so that more complex applications can be constructed from these manageable units. Parallel PDE-based applications are unique in this context only to the extent that they tend to be extremely complex and thus can profoundly benefit from component concepts. The CCA contributes a component model for high-performance scientific computing that may be of particular interest to investigators conducting simulations of detailed or comprehensive physical phenomena. Compliance with the CCA specification has enabled the scientific application teams featured in this chapter to perform several tasks more easily:

- Create sets of reusable, easy-to-maintain, and scalable components, each of which expresses a unique physical or numerical functionality [72,77,79,101,139]
- Use legacy code (originally written in Fortran or C) in the CCA environment without major code rewrites [72, 77]
- Easily test different physics and numerics modules with well-defined interfaces in a plug-and-play mode [79]

- Manage the evolution of complex scientific applications, by separating the disparate concerns of physics, numerical, and computer science issues [39, 101]
- Obtain good parallel performance [40,79,87] with negligible CCA overhead [20]

There is no point at which we envision the CCA as a component model will be finished. The CCA continues to respond to implementers' concerns, feature requests, and unforeseen conflicts created by CCA-specified mechanisms or the lack thereof. In addition, the CCA Forum is extending the prototype work of Section 4 and assembling a critical mass of components from which parallel simulations can be prototyped and evolved into meaningful simulations. Component concepts also provide unprecedented opportunities for automation. Recent work on *computational quality of service* [59,96] allows component parameters and component configurations to be rearranged dynamically, thereby enabling the automatic selection and configuration of components to suit the computational conditions imposed by a simulation and its operating environment. The CCA Forum aims to enable next-generation high-performance scientific simulations by providing a means for tens or even hundreds of researchers to contribute to a single application as well as by developing the infrastructure to automate its construction and execution.

Acknowledgments

The CCA has been under development since 1998 by the CCA Forum and represents the contributions of many people, all of whom we gratefully acknowledge. We also thank our collaborators outside the CCA Forum, especially the domain scientists who have contributed to the four applications discussed in this chapter and the early adopters of the CCA, for the important contributions they have made both to our understanding of CBSE in the high-performance scientific computing context and to making the CCA a practical and usable environment. We thank Barry Smith for developing a PETSc-based implementation of the TOPS solver interfaces discussed in Section 4.4. In addition, we thank Are Magnus Bruaset, Jeff Keasler, Barry Smith, and the anonymous reviewers of this chapter for valuable feedback that has enabled us to improve our presentation.

This work has been supported in part by the U. S. Department of Energy's Scientific Discovery through Advanced Computing (SciDAC) [125] initiative, through the Center for Component Technology for Terascale Simulation Software, of which Argonne, Lawrence Livermore, Los Alamos, Oak Ridge, Pacific Northwest, and Sandia National Laboratories, Indiana University, and the University of Utah are members. Members of the SciDAC Computational Facility for Reacting Flow Research have also contributed to this paper.

Research at Argonne National Laboratory was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

Some of this work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Oak Ridge National Laboratory is managed by UT-Battelle, LLC for the US Dept. of Energy under contract DE-AC-05-00OR22725.

This research was performed in part using the Molecular Science Computing Facility (MSCF) in the William R. Wiley Environmental Laboratory at the Pacific Northwest National Laboratory (PNNL). The MSCF is funded by the Office of Biological and Environmental Research in the U.S. Department of Energy. PNNL is operated by Battelle for the U.S. Department of Energy under contract DE-AC06-76RLO 1830.

Research at the University of Utah is also sponsored by the National Science Foundation under contract ACI0113829, and the DOE ASC Program.

Some of the work in this paper was carried out by Northrop Grumman with funding provided by NASA's Computation Technologies (CT) Project, part of the Earth Science Technology Office (ESTO), under a contract with the National Aeronautics and Space Administration.

References

1. Ahrem, R., Post, P., Steckel, B., and Wolf, K.: MpCCI: A tool for coupling CFD with other disciplines. In *Proceedings of the Fifth World Conference in Applied Fluid Dynamics, CFD-Efficiency and Economic Benefit in Manufacturing*, (2001)
2. Ahrem, R., Post, P., and Wolf, K.: A communication library to couple simulation codes on distributed systems for multi-physics computations. In D'Hollander, E., Joubert, G., Peters, F., and Sips, H., editors, *Parallel Computing: Fundamentals and Applications, Proceedings of the International Conference ParCO 99*, pages 47–55. Imperial College Press, (1999)
3. Allan, B., Armstrong, R., Lefantzi, S., Ray, J., Walsh, E., and Wolfe, P.: Ccaffeine – a CCA component framework for parallel computing. `http://www.cca-forum.org/ccafe/`, (2003)
4. Allan, B. A., Armstrong, R. C., Wolfe, A. P., Ray, J., Bernholdt, D. E., and Kohl, J. A.: The CCA core specification in a distributed memory SPMD framework. *Concurrency and Computation: Practice and Experience*, 14(5):1–23, (2002)
5. Allan, B. A., Lefantzi, S., and Ray, J.: ODEPACK++: Refactoring the LSODE Fortran library for use in the CCA high performance component software architecture. In *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, Santa Fe, NM, (2004). IEEE Press. see `http://www.cca-forum.org/~baallan/odepp`
6. Allen, G., Benger, W., Goodale, T., Hege, H., Lanfermann, G., Merzky, A., Radke, T., Seidel, E., and Shalf, J.: The Cactus code: A problem solving environment for the Grid. In *High Performance Distributed Computing (HPDC)*, pages 253–260. IEEE Computer Society, (2000)
7. Alpatov, P., Baker, G., Edwards, C., Gunnels, J., Morrow, G., Overfelt, J., van de Geijn, R., and Wu, Y.-J. J.: PLAPACK: Parallel linear algebra package - design overview. In *Proceedings of SC97*, (1997)

8. Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., and Smolinski, B.: Toward a Common Component Architecture for high-performance scientific computing. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, (1999)
9. B. P. Sommeijer, L. F. S. and Verwer, J. G.: RKC: an explicit solver for parabolic PDEs. *J. Comp. Appl. Math.*, 88:315–326, (1998)
10. Balay, S., Buschelman, K., Gropp, W., Kaushik, D., Knepley, M., McInnes, L., Smith, B. F., and Zhang, H.: PETSc users manual. Technical Report ANL-95/11 - Revision 2.2.1, Argonne National Laboratory, (2004). `http://www.mcs.anl.gov/petsc`
11. Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F.: Efficient management of parallelism in object oriented numerical software libraries. In Arge, E., Bruaset, A. M., and Langtangen, H. P., editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, (1997)
12. Beckman, P., Fasel, P., Humphrey, W., and Mniszewski, S.: Efficient coupling of parallel applications using PAWS. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computation*, (1998)
13. Benson, S., Krishnan, M., McInnes, L., Nieplocha, J., and Sarich, J.: Using the GA and TAO toolkits for solving large-scale optimization problems on parallel computers. Technical Report ANL/MCS-P1084-0903, Argonne National Laboratory, (2003)
14. Benson, S., McInnes, L. C., and Moré, J.: A case study in the performance and scalability of optimization algorithms. *ACM Transactions on Mathematical Software*, 27:361–376, (2001)
15. Benson, S., McInnes, L. C., Moré, J., and Sarich, J.: TAO users manual. Technical Report ANL/MCS-TM-242 - Revision 1.7, Argonne National Laboratory, (2004). `http://www.mcs.anl.gov/tao/`
16. Benson, S. and Moré, J.: A limited-memory variable-metric algorithm for bound-constrained minimization. Technical Report ANL/MCS-P909-0901, Mathematics and Computer Science Division, Argonne National Laboratory, (2001)
17. Berger, M. J. and Oliger, J.: Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comp. Phys.*, 53:484–523, (1984)
18. Berger, M. and Colella, P.: Local adaptive mesh refinement for shock hydrodynamics. *J. Comp. Phys.*, 82:64–84, (1989)
19. Bernholdt, D. E., Armstrong, R. C., and Allan, B. A.: Managing complexity in modern high end scientific computing through component-based software engineering. In *Proceedings. of the HPCA Workshop on Productivity and Performance in High-End Computing (P-PHEC 2004), Madrid, Spain*. IEEE Computer Society, (2004)
20. Bernholdt, D. E., Elwasif, W. R., Kohl, J. A., and Epperly, T. G. W.: A component architecture for high-performance computing. In *Proceedings of the Workshop on Performance Optimization via High-Level Languages and Libraries (POHLL-02)*, (2002)
21. Bernholdt, D. E., Allan, B. A., Armstrong, R., Bertrand, F., Chiu, K., Dahlgren, T. L., Damevski, K., Elwasif, W. R., Epperly, T. G. W., Govindaraju, M., Katz, D. S., Kohl, J. A., Krishnan, M., Kumfert, G., Larson, J. W., Lefantzi, S., Lewis, M. J., Malony, A. D., McInnes, L. C., Nieplocha, J., Norris, B., Parker, S. G., Ray, J., Shende, S., Windus, T. L., and Zhou, S.: A component architecture for high-performance scientific computing, (2004). submitted to *Intl. J. High-Perf. Computing Appl.*
22. Bertrand, F., Bramley, R., Damevski, K., Kohl, J., Larson, J., and Sussman, A.: MxN interactions in parallel component architectures. Technical Report TR604, Department of Computer Science, Indiana University, Bloomington, (2004). Accepted by the International Parallel and Distributed Processing Symposium, Denver, CO, April 4-8, 2005

23. Bettge, T., Craig, A., James, R., and Wayland, V.: The DOE Parallel Climate Model (PCM): The computational highway and backroads. In Alexandrov, V. N., Dongarra, J. J., Juliano, B. A., Renner, R. S., and Tan, C. J. K., editors, *Proceedings of the International Conference on Computational Science (ICCS) 2001*, volume 2073 of *Lecture Notes in Computer Science*, pages 148–156, Berlin, (2001). Springer-Verlag
24. Blelloch, G. E., Heroux, M. A., and Zagha, M.: Segmented operations for sparse matrix computation on vector multiprocessor. Technical Report CMU-CS-93-173, Carnegie Mellon University, (1993)
25. Boost. `http://www.boost.org`, (2004)
26. Box, D.: *Essential COM*. Addison-Wesley, (1997)
27. Bramley, R., Gannon, D., Stuckey, T., Vilacis, J., Akman, E., Balasubramanian, J., Berg, F., Diwan, S., and Govindaraju, M.: The linear system analyzer. In *Enabling Technologies for Computational Science*, Kluwer, 2000
28. CCA Forum homepage. `http://www.cca-forum.org/`, (2004)
29. CCA specification. `http://cca-forum.org/specification/`, (2004)
30. Chatterjee, S., Blelloch, G. E., and Zagha, M.: Scan primitives for vector computers. In *Supercomputing 1990*, (1990)
31. Clay, R. et al.: ESI homepage. `http://www.terascale.net/esi`, (2001)
32. Cockburn, B. and Shu, C.-W.: The local discontinuous Galerkin method for time-dependent convection-diffusion systems. *SIAM J. Numer. Anal.*, 35(6):2440–2463, (1998)
33. Cohen, S. D. and Hindmarsh, A. C.: CVODE, a stiff/nonstiff ODE solver in C. *Computers in Physics*, 10(2):138–143, (1996)
34. Colella, P.: An Algorithmic and Software Framework for Applied Partial Differential Equations Center (APDEC). `http://davis.lbl.gov/APDEC/`, (2004)
35. Colella, P. et al.: Chombo – Infrastructure for Adaptive Mesh Refinement. `http://seesar.lbl.gov/anag/chombo`
36. Colorado State University: The CSU GCM (BUGS) homepage. `http://kiwi.atmos.colostate.edu/BUGS/`, (2004)
37. Combustion Research Facility. `http://www.ca.sandia.gov/CRF`, (2004)
38. Dahlgren, T., Epperly, T., and Kumfert, G.: *Babel User's Guide*. CASC, Lawrence Livermore National Laboratory, version 0.9.0 edition, (2004)
39. de St. Germain, J. D., McCorquodale, J., Parker, S. G., and Johnson, C. R.: Uintah: A massively parallel prolem solving environment. In *Proceedings of the Ninth IEEE International Symposium on High Performance and Distributed Computing*, (August 2000)
40. de St. Germain, J., Morris, A., Parker, S., Malony, A., and Shende, S.: Integrating performance analysis in the Uintah software development cycle. In *The Fourth International Symposium on HighPerformance Computing (ISHPC-IV)*, pages 190–206, (2002)
41. Durran, D. R.: *Numerical Methods for Wave Equations in Geophysical Fluid Dynamics*. Springer, (1999)
42. Edjlali, G., Sussman, A., and Saltz, J.: Interoperability of data-parallel runtime libraries. In *International Parallel Processing Symposium*, Geneva, Switzerland, (1997). IEEE Computer Society Press
43. Eisenhauer, G., Bustamante, F., and Schwan, K.: Event services for high performance systems. *Cluster Computing: The Journal of Networks, Software Tools, and Applications*, 3(3), (2001)
44. Englander, R.: *Developing Java Beans*. O'Reilly and Associates, (1997)
45. Feo, J. T., Cann, D. C., and Oldehoeft, R. R.: A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, (1990)

46. Gamma, E. et al.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, (1994)
47. Ge, L., Lee, L., Zenghai, L., Ng, C., Ko, K., Luo, Y., and Shephard, M.: Adaptive mesh refinement for high accuracy wall loss determination in accelerating cavity design. In *IEEE Conf. on Electromagnetic Field Computations*, (2004)
48. Geist, G. A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V.: *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, (1994)
49. Geist, G. A., Kohl, J. A., and Papadopoulos, P. M.: CUMULVS: Providing fault tolerance, visualization and steering of parallel applications. *Intl. J. High-Perf. Computing Appl.*, 11(3):224–236, (1997)
50. GFDL Flexible Modeling System. `http://www.gfdl.noaa.gov/fms`, (2004)
51. Gockenbach, M. S., Petro, M. J., and Symes, W. W.: C++ classes for linking optimization with complex simulations. *ACM Transactions on Mathematical Software*, 25(2):191–212, (1999)
52. Govindaraju, M., Krishnan, S., Chiu, K., Slominski, A., Gannon, D., and Bramley, R.: Merging the CCA component model with the OGSI framework. In *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, (2003)
53. Guilyardi, E., Budich, R. G., and Valcke, S.: PRISM and ENES: European approaches to Earth System Modelling. In *Proceedings of Realizing TeraComputing - Tenth Workshop on the Use of High Performance Computing in Meteorology*, (2002)
54. Harper, L. and Kauffman, B.: Community Climate System Model. `http://www.ccsm.ucar.edu/`, (2004)
55. Henderson, T. C., McMurtry, P. A., Smith, P. J., Voth, G. A., Wight, C. A., and Pershing, D. W.: Simulating accidental fires and explosions. *Comp. Sci. Eng.*, 2:64–76, (1994)
56. Heroux, M. A. and Willenbring, J. M.: Trilinos Users Guide. Technical Report SAND2003-2952, Sandia National Laboratories, (2003). `http://software.sandia.gov/Trilinos`
57. Hill, C. et al.: The architecture of the earth system modeling framework. *Computing in Science and Engineering*, 6(1):18–28, (2003)
58. Hindmarsh, A. C.: ODEPACK, a systematized collection of ODE solvers. *Scientific Computing*, (1993)
59. Hovland, P., Keahey, K., McInnes, L. C., Norris, B., Diachin, L. F., and Raghavan, P.: A quality of service approach for high-performance numerical components. In *Proceedings of Workshop on QoS in Component-Based Software Engineering, Software Technologies Conference*, Toulouse, France, (2003)
60. Hunke, E. C. and Dukowicz, J. K.: An elastic-viscous-plastic model for sea ice dynamics. *J. Phys. Oc.*, 27:1849–1867, (1997)
61. Indiana University: XCAT homepage. `http://www.extreme.indiana.edu/xcat/`
62. Karniadakis, G. E. and Sherwin, S. J.: *Spectral/Hp Element Methods for CFD*. Numerical Mathematics and Scientific Computation. Oxford University Press, (1999)
63. Keahey, K., Fasel, P., and Mniszewski, S.: PAWS: Collective interactions and data transfers. In *Proceedings of the High Performance Distributed Computing Conference*, San Francisco, CA, (2001)
64. Keahey, K., Beckman, P., and Ahrens, J.: Ligature: Component architecture for high performance applications. *Intl. J. High-Perf. Computing Appl.*, 14(4):347–356, (2000)
65. Kenny, J. P., Benson, S. J., Alexeev, Y., Sarich, J., Janssen, C. L., McInnes, L. C., Krishnan, M., Nieplocha, J., Jurrus, E., Fahlstrom, C., and Windus, T. L.: Component-based

integration of chemistry and optimization software. *J. of Computational Chemistry*, 25(14):1717–1725, (2004)
66. Keyes, D.: Terascale Optimal PDE Simulations (TOPS) Center. `http://tops-scidac.org/`, (2004)
67. Killeen, T., Marshall, J., and da Silva, A.: Earth System Modeling Framework. `http://www.esmf.ucar.edu`, (2004)
68. Knio, O., Najm, H., and Wyckoff, P.: A semi-implicit numerical scheme for reacting flow. II. stiff, operator-split formulation. *J. Comp. Phys.*, 154:428–467, (1999)
69. Kohl, J. A. and Papadopoulos, P. M.: A library for visualization and steering of distributed simulations using PVM and AVS. In *High Performance Computing Symposium*, Montreal, CA, (1995)
70. Lam, S. H. and Goussis, D. A.: The CSP method of simplifying kinetics. *International Journal of Chemical Kinetics*, 26:461–486, (1994)
71. Larson, J. W., Jacob, R. L., Foster, I. T., and Guo, J.: The Model Coupling Toolkit. In Alexandrov, V. N., Dongarra, J. J., Juliano, B. A., Renner, R. S., and Tan, C. J. K., editors, *Proceedings of the International Conference on Computational Science (ICCS) 2001*, volume 2073 of *Lecture Notes in Computer Science*, pages 185–194, Berlin, (2001). Springer-Verlag
72. Larson, J. W., Norris, B., Ong, E. T., Bernholdt, D. E., Drake, J. B., Elwasif, W. R., Ham, M. W., Rasmussen, C. E., Kumfert, G., Katz, D. S., Zhou, S., DeLuca, C., and Collins, N. S.: Components, the Common Component Architecture, and the climate/weather/ocean community. In *84th American Meteorological Society Annual Meeting*, Seattle, Washington, (2004). American Meteorological Society
73. Larson, J., Jacob, R., and Ong, E.: The Model Coupling Toolkit: A new Fortran90 toolkit for building multi-physics parallel coupled models. Technical Report ANL/MCS-P1208-1204, Argonne National Laboratory, (2004). Submitted to Int. J. High Perf. Comp. App. See also `http://www.mcs.anl.gov/mct/`
74. Lawrence Livermore National Laboratory: Babel. `http://www.llnl.gov/CASC/components/babel.html`, (2004)
75. Lee, J. C., Najm, H. N., Valorani, M., and Goussis, D. A.: Using computational singular perturbation to analyze large scale reactive flows. In *Proceedings of the Fall Meeting of the Western States Section of the The Combustion Institute*, Los Angeles, California, (2003). Distributed via CD-ROM
76. Lefantzi, S., Kennedy, C., Ray, J., and Najm, H.: A study of the effect of higher order spatial discretizations in SAMR (Structured Adaptive Mesh Refinement) simulations. In *Proceedings of the Fall Meeting of the Western States Section of the The Combustion Institute*, Los Angeles, California, (2003). Distributed via CD-ROM
77. Lefantzi, S. and Ray, J.: A component-based scientific toolkit for reacting flows. In *Proceedings of the Second MIT Conference on Computational Fluid and Solid Mechanics, June 17-20, 2003, Cambridge, MA*, volume 2, pages 1401–1405. Elsevier, (2003)
78. Lefantzi, S., Ray, J., Kennedy, C., and Najm, H.: A component-based toolkit for reacting flow with high order spatial discretizations on structured adaptively refined meshes. *Progress in Computational Fluid Dynamics: An International Journal*, (2004). To appear
79. Lefantzi, S., Ray, J., and Najm, H. N.: Using the Common Component Architecture to design high performance scientific simulation codes. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France*. IEEE Computer Society, (2003)

80. Lefantzi, S., Ray, J., and Shende, S.: Strong scalability analysis and performance evaluation of a CCA-based hydrodynamic simulation on structured adaptively refined meshes. Poster in ACM/IEEE Conference on Supercomputing, November 2003, Phoenix, AZ
81. Lele, S.: Compact finite differnece schemes with spectral-like resolution. *J. Comp. Phys.*, 103:16–42, (1992)
82. Lilek, Z. and Perić, M.: A fourth-order finite volume method with collocated variable arrangement. *Computers & Fluids*, 24, (1995)
83. Lin, S. J. et al.: Global weather prediction and high-end computing at NASA. *Computing in Science and Engineering*, 6(1):29–35, (2003)
84. Lindemann, J., Dahlblom, O., and Sandberg, G.: Using CORBA middleware in finite element software. In Sloot, P. M. A., Tan, C. J. K., Dongarra, J. J., , and Hoekstra, A. G., editors, *Proceedings of the 2nd International Conference on Computational Science*, Lecture Notes in Computer Science. Springer, (2002). To appear in *Future Generation Computer Systems* (2004).
85. Lumsdaine, A. et al.: Matrix Template Library. `http://www.osl.iu.edu/research/mtl`, (2004)
86. Massachussetts Institute of Technology: The MIT GCM homepage. `http://mitgcm.org/`, (2004)
87. McCorquodale, J., de St. Germain, J., Parker, S., and Johnson, C.: The Uintah parallelism infrastructure: A performance evaluation on the SGI Origin 2000. In *High Performance Computing 2001*, (2001)
88. Microsoft Corporation: Component Object Model specification. `http://www.microsoft.com/com/resources/comdocs.asp`, (1999)
89. Microsoft Corporation: Distributed Component Object Model. `http://www.microsoft.com/com/tech/dcom.asp`, (2004)
90. Moré, J. J. and Wright, S. J.: *Optimization Software Guide*. SIAM Publications, Philadelphia, (1993)
91. MPI Forum: MPI: a message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4):159–416, (1994)
92. Najm, H. N., Schefer, R. W., Milne, R. B., Mueller, C. J., Devine, K. D., and Kempka, S. N.: Numerical and experimental investigation of vortical flow-flame interaction. SAND Report SAND98-8232, UC-1409, Sandia National Laboratories, Livermore, CA 94551-0969, (1998)
93. Najm, H. N. et al.: CFRFS homepage. `http://cfrfs.ca.sandia.gov/`, (2003)
94. Nieplocha, J., Harrison, R. J., and Littlefield, R. J.: Global arrays: A non-uniform-memory-access programming model for high-performance computers. *J. Supercomputing*, 10(2):169, (1996)
95. Norris, B., Balay, S., Benson, S., Freitag, L., Hovland, P., McInnes, L., and Smith, B.: Parallel components for PDEs and optimization: Some issues and experiences. *Parallel Computing*, 28(12):1811–1831, (2002)
96. Norris, B., Ray, J., Armstrong, R., McInnes, L. C., Bernholdt, D. E., Elwasif, W. R., Malony, A. D., and Shende, S.: Computational quality of service for scientific components. In *Proc. of International Symposium on Component-Based Software Engineering (CBSE7), Edinburgh, Scotland*, (2004)
97. Object Management Group: CORBA component model. `http://www.omg.org/technology/documents/formal/components.htm`, (2002)
98. Pacific Northwest National Laboratory: Global Array Toolkit homepage. `http://www.emsl.pnl.gov:2080/docs/global/`, (2004)

99. Palmer, B. and Nieplocha, J.: Efficient algorithms for ghost cell updates on two classes of MPP architectures. In *14th IASTED International Conference on Parallel and Distributed Computing and Systems*, Cambridge, MA, (2002)
100. Parashar, M. et al.: GrACE homepage. `http://www.caip.rutgers.edu/~parashar/TASSL/Projects/GrACE/`, (2004)
101. Parker, S. G.: A component-based architecture for parallel multi-physics PDE simulation. In *Proceedings of the International Conference on Computational Science-Part III*, pages 719–734. Springer-Verlag, (2002)
102. Pérez, C., Priol, T., and Ribes, A.: A parallel CORBA component model for numerical code coupling. *Intl. J. High-Perf. Computing Appl.*, 17(4), (2003)
103. Peyret, R. and Taylor, T.: *Computational Methods for Fluid Flow*, chapter 6. Springer Series in Computational Physics. Springer-Verlag, New York, (1983). Finite-Difference Solution of the Navier-Stokes Equations
104. Phillip Jones: Parallel Ocean Program (POP) homepage. `http://climate.lanl.gov/Models/POP/`, (2004)
105. Poinsot, T., Candel, S., and Trouvé, A.: Applications of direct numerical simulation to premixed turbulent combustion. *Progress in Energy and Combustion Science*, 21:531–576, (1995)
106. Pozo, R.: Template Numerical Toolkit. `http://math.nist.gov/tnt`, (2004)
107. Radhakrishnan, K. and Hindmarsh, A. C.: Description and use of LSODE, the Livermore solver for ordinary differential equations. Technical Report UCRL-ID-113855, Lawrence Livermore National Laboratory, (1993)
108. Ranganathan, M., Acharya, A., Edjlali, G., Sussman, A., and Saltz, J.: A runtime coupling of data-parallel programs. In *Proceedings of the 1996 International Conference on Supercomputing*, Philadelphia, PA, (1996)
109. Ray, J., Allan, B. A., Armstrong, R., and Kohl, J.: Structured mesh demo for supercomputing 2004. `http://www.cca-forum.org/~jaray/SC04/sc04.html`, (2004)
110. Ray, J., Kennedy, C., Lefantzi, S., and Najm, H.: High-order spatial discretizations and extended stability methods for reacting flows on structured adaptively refined meshes. In *Proceedings of the Third Joint Meeting of the U.S. Sections of The Combustion Institute, March 16-19, 2003, Chicago, Illinois.*, (2003). Distributed via CD-ROM
111. Reynders, J. V. W., Cummings, J. C., Hinker, P. J., Tholburn, M., S. Banerjee, M. S., Karmesin, S., Atlas, S., Keahey, K., and Humphrey, W. F.: *POOMA: A FrameWork for Scientific Computing Applications on Parallel Architectures*, chapter 14. MIT Press, (1996)
112. Roman, E.: *Mastering Enterprise JavaBeans*. O'Reilly and Associates, (1997)
113. Sarich, J.: A programmer's guide for providing CCA component interfaces to the Toolkit for Advanced Optimization. Technical Report ANL/MCS-TM-279, Argonne National Laboratory, (2004)
114. Smith, B. et al.: TOPS Solver Interface. `http://www-unix.mcs.anl.gov/scidac-tops/tops-solver-interface`, (2004)
115. Smith, K., Ray, J., and Allan, B. A.: CVODE component user guidelines. Technical Report SAND2003-8276, Sandia National Laboratory, (2003)
116. Sportisse, B.: An analysis of operator splitting techniques in the stiff case. *J. Comp. Phys.*, 161:140–168, (2000)
117. Strang, G.: On the construction and comparison of difference schemes. *SIAM J. Numer. Anal.*, 5(3):506–517, (1968)
118. Suarez, M. J. and Takacs, L.: Documentation of the Aries-GEOS dynamical core: Version 2. Technical Report TM-1995-104606, NASA, (1995)

119. Sulsky, D., Chen, Z., and Schreyer, H. L.: A Particle Method for History Dependent Materials. *Comp . Methods Appl. Mech. Engrg*, 118, (1994)
120. Sun, Y., Folwell, N., Li, Z., and Golub, G.: High precision accelerator cavity design using the parallel eigensolver Omega3P. In *Proc. of the 18th Annual Review of Progress in Applied Computational Electromagnetics ACES 2002*, Monterey, CA, (2002)
121. Szyperski, C.: *Component Software: Beyond Object-Oriented Programming*. ACM Press, New York, (1999)
122. Talbot, B., Zhou, S., and Higgins, G.: Software engineering support of the third round of scientific grand challenge investigations–earth system modeling software framework survey task4 report. Technical Report TM-2001-209992, NASA, (2001)
123. Trease, H.E.and Trease, L.: NWGrid: A multi-dimensional, hybrid, unstructured, parallel mesh generation system. `http://www.emsl.pnl.gov/nwgrid`, (2000)
124. The Terascale Simulation Tools and Technologies (TSTT) Center. `http://www.tstt-scidac.org`, (2004)
125. U. S. Dept. of Energy: SciDAC Initiative homepage. `http://www.osti.gov/scidac/`, (2003)
126. University Corporation for Atmospheric Research: The Community Atmosphere Model (CAM) homepage. `http://www.ccsm.ucar.edu/models/atm-cam/`, (2004)
127. University of Oregon: TAU: Tuning and analysis utilities. `http://www.cs.uoregon.edu/research/paracomp/tau`, (2003)
128. Vajracharya, S., Karmesin, S., Beckman, P., Crotinger, J., Malony, A., Shende, S., Oldehoeft, R., and Smith, S.: Smarts: Exploiting temporal locality and parallelism through vertical execution. In *Proceedings of the 13th International Conference on Supercomputing (ICS 99)*, pages 302–310, Rhodes, Greece, (1999). ACM Press
129. Veldhuizen, T. et al.: BLITZ++: Object-oriented scientific computing. `http://www.oonumerics.org/blitz`, (2004)
130. Visbal, M. R. and Gaitonde, D. V.: On the use of higher-order finite-difference schemes on curvilinear and deforming meshes. *J. Comp. Phys.*, 181:155–185, (2002)
131. Wang, Z. and Huang, G. P.: An essentially nonoscillatory high-order Padé-type (ENO-Padé) scheme. *J. Comp. Phys.*, 177:37–58, (2002)
132. Weather Research and Forecasting Model. `http://www.wrf-model.org/`, (2004)
133. Williams, F.: *Combustion Theory*. Addison-Wesley, New York, 2nd edition, (1985)
134. Wissink, A., Hornung, R., Kohn, S., Smith, S., and Elliott, N.: Large scale parallel structured AMR calculations using the SAMRAI framework. In *Proceedings of the SC01 Conf. High Perf. Network. and Comput*, Denver, CO, (2001)
135. Wissink, A., Hysom, D., and Hornung, R.: Enhancing scalability of parallel structured AMR calculations. In *Proceedings of the* 17^{th} *ACM International Conference on Supercomputing (ICS03)*, pages 336–347, San Francisco, CA, (2003)
136. Wolf, M., Cai, Z., Huang, W., and Schwan, K.: Smart pointers: Personalized scientific data portals in your hand. In *Proceedings of Supercomputing 2002*, (2002)
137. Wolf, M., Guetz, A., and Ng, C.-K.: Modeling large accelerator structures with the parallel field solver Tau3P. In *Proc. of the 18th Annual Review of Progress in Applied Computational Electromagnetics ACES 2002*, Monterey, CA, (2002)
138. Zhang, K., Damevski, K., Venkatachalapathy, V., and Parker, S.: SCIRun2: A CCA framework for high performance computing. In *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, Santa Fe, NM, (2004). IEEE Press

139. Zhou, S., da Silva, A., Womack, B., and Higgins, G.: Prototyping the ESMF using DOE's CCA. In *NASA Earth Science Technology Conference 2003*, College Park, MD, (2003). `http://esto.nasa.gov/conferences/estc2003/papers/A4P3(Zhou).pdf`
140. Zhou, S.: Coupling earth system models: An ESMF-CCA prototype. `http://webserv.gsfc.nasa.gov/ESS/esmf_tasc`, (2003)