Computational frameworks for advanced combustion simulations

J. Ray, R. Armstrong, C. Safta, B. J Debusschere, B. A. Allan and H. N. Najm

Abstract Computational frameworks can significantly assist in the construction, extension and maintenance of simulation codes. As the nature of problems addressed by computational means has grown in complexity, such frameworks have evolved to incorporate a commensurate degree of sophistication, both in terms of the numerical algorithms that they accommodate as well as the software architectural discipline they impose on their users. In this chapter, we discuss a *component framework*, the Common Component Architecture (CCA), for developing scientific software, and describe how it has been used to develop a toolkit for simulating reacting flows. In particular, we will discuss why a component architecture was chosen and the philosophy behind the particular software design. Using statistics drawn from the toolkit, we will analyze the code structure and investigate to what degree the aims of the software design were actually realized. We will explore how CCA was employed to design a high-order simulation code on block-structured adaptive meshes, as well as a simulation capacity for adaptive stiffness reduction in detailed chemical models. We conclude the chapter with two reacting flow studies performed using the above-mentioned computational capabilities.

1 Introduction

Computational science has come to regarded as the third leg of science, after theory and experimentation. With the advent of massively parallel computers, simulations have been used to investigate extremely challenging problems. However, as the problems have increased in complexity, the tools used to investigate them computationally – numerical algorithms and their software implementations – have developed a commensurate sophistication and intricacy. Software complexity, with

J. Ray, R. Armstrong, C. Safta, B. Debusschere, B. A. Allan and H. N. Najm
Sandia National Laboratories, Livermore, CA, e-mail: [jairay,rob,csafta,bjdebus,baallan,hnnajm]@sandia.gov

its detrimental impact on software maintainability and extensibility, is regarded as a large drain on time and effort and computational frameworks are intended to be a solution to this problem. One means of addressing complexity is to provide "shrink-wrapped" functionality, in essence transferring software and algorithmic complexity to the implementer of the framework from its user. A second way of addressing complexity is through some form of modularization. *Component frameworks* fall in the second category.

The work "framework" is an extremely overused and consequently, confusing, term. It can refer to an *architecture*, a set of specifications that, when followed, imposes some standard on the software that adhere to it. By this token, a component framework is a set of specifications that permit software to be constructed by modular composition. The modules are referred to as *components*. Confusingly, "framework" can also refer to a software framework that is written to such a specification and is meant to locate, instantiate and compose components. The aim of component frameworks is to use *modularity* to divide and conquer complexity, by composing programs out of software building blocks. Studies have shown that the capability and complexity of a program is proportional to its size, which determines the number of software developers required to maintain the software [63, 36]. Component frameworks can be used to design an organizing principle for the efficient deployment of programming resources. Individual programmers have cognizance over one or more components of the overall program, which provides a natural division of responsibilities. It makes obvious that the interfaces between components are where programmers must negotiate. Interfaces exported to, or imported from components is functionality that is needed by, or provided to a programmer as part of his/her domain of responsibility. The structure provided by component-based software engineering is one of the main reasons for using it and defines the workflow in a large software project.

In this chapter, we will limit the discussion to the Common Component Architecture (CCA), a set of specifications for developing and composing component-based scientific simulation software. We will devote some time to the CCA frameworks used in scientific computings and provide examples of how CCA has been used for software for the simulation of flames.

2 Literature review of computational frameworks

Scientific computational frameworks proliferate. Some were initially designed to assist in the creation of simulation codes in a certain scientific field and consequently have been adopted/adapted rather sporadically elsewhere. Others were intended to assist in nothing more than the implementation of a particular software design, irrespective of the nature/aim of the software being developed. It is impossible to review the myriads of scientific simulation frameworks that lie between these two extremes and this will not be attempted here. Since this chapter describes the design of a component-based simulation facility for reacting flows on adaptively refined

meshes, we will devote the majority of our review to frameworks that assist in computations on block-structured adaptively refined meshes, and implementations of CCA that are relevant to scientific computing. We will, however, describe a basic categorization before proceeding with the details.

Scientific computing frameworks are application development frameworks which, at the very least, support parallel computing and pay particular attention to high performance in their design. This manifests itself in their ability to address operations on large arrays and the elimination/reduction of overheads that scale with the size of the data or with the number of processors on which the framework is expected to run. Further, they rarely contain any support for languages outside FORTRAN and C/C++, and make sparing use of remote method invocation. Within these confines, scientific simulation frameworks can be divided into three categories.

In the first category are frameworks that were originally designed to support a particular field of science. The Earth System Modeling Framework [30, 32] is one such example, developed for constructing climate simulation codes; Cactus [37, 15] is another, developed for numerical relativity studies while FLASH [31, 35], developed for astrophysical simulation, forms a third example. OpenFoam, developed initially for finite-volume simulation of fluid flow (but now containing extensions for solid mechanics and Direct Simulation Monte Carlo calculations as well) [80] is in widespread use. Such application-specific frameworks often contain numerical schemes and "utility" scientific models that find routine use. They are modular in design and implement many of the ideas (e.g., involving a separation of implementation from the interface of a module) that are formalized in CCA. Note that these frameworks do not intend to impose any design patterns on the resulting simulation code; rather one adopts the data structures and design patterns of the framework itself and reaps the benefits of using validated solvers and models in one's simulations. These frameworks allow one to rapidly develop simulation codes as long as the facilities provided by the framework (e.g. solvers and models) are profitably leveraged by the simulation.

The second category of frameworks consists of those that enable the use of a particular solution methodology e.g.. block-structured adaptive mesh refinement (AMR) in simulation codes. These frameworks are more general that those described in the previous paragraph – they provide data-structures and solvers (or interfaces to those implemented by external libraries) that are useful while developing simulation software. Again, they make no attempt to promote any particular design pattern. However, given that one makes heavy use of the framework-provided data structures (which are usually implemented in the form of objects), the design patterns employed for simulation codes bear a strong resemblance to those employed for the framework itself. We will review a few such frameworks/packages that enable the use of AMR in simulation codes.

Simulations using block-structured adaptively refined meshes are generally conducted where the spatial domain is rectangular/cuboid or can be logically described as such. One begins with a Cartesian mesh with a resolution that is insufficient to capture many of the features of the solution. Regions in the domain that require refinement are identified and collated into rectangles/cuboids and a finer mesh (usu-

ally, finer by a constant factor) is overlaid on the refined part of the initial mesh. This is performed recursively leading to a *mesh hierarchy* of a few (usually less than 10) levels. GrACE [67, 38], AMROC [25, 70, 5], CHOMBO [17, 69] and AmrLib [4] are a few frameworks that implement such an adaptive meshing strategy. These frameworks provide the infrastructure required to manipulate such meshes as well as data structures that allow fields to be described on them (on distributed memory computers, this involves very intricate book-keeping). The data structures also allow the use of *time-refinement* [13], a time-integration technique that allows one to mitigate the effect of having time-steps be CFL-constrained by the finest mesh in a mesh-hierarchy. While CHOMBO stores the data on a given level of the mesh hierarchy in a separate data-structure, both AMROC and GrACE present a data object that is described on the entire hierarchy. The three frameworks also largely automate the work of refining/coarsening the mesh hierarchy periodically (called regridding), based on a user-calculated "error metric" and perform load-balancing of the mesh and data after each regridding operation. While GrACE only provides a parallel block-structured adaptive mesh and the associated data object, CHOMBO and AMROC also provide a set of commonly used solvers.

All the three frameworks have been used for simulating reacting flows. AMROC has its origins in the simulation of shock-laden flows [26, 66], but has been extended to shock-fluid interactions [29, 18, 28] and shock-induced combustion [27]. CHOMBO (and its predecessors) have been used for an immense variety of simulations [69], including solutions of variable density formulation of the Navier-Stokes equations [3], embedded boundary methods [20], and fourth-order-in-space discretizations [21, 9] for AMR simulations. AmrLib, which has similar foundations as CHOMBO, has been used to develop algorithms for low Mach laminar flames simulations [24] as well as for turbulent flames [10, 11]. GrACE was initially developed to investigate load-balancers for block-structured adaptive meshes [54, 55], but has been used to develop efficient numerical schemes (e.g., extended stability time integrators [53] and fourth-order spatial discretizations [73]) for the simulation of flames [75]. These frameworks have been investigated for their scalability on parallel machines; see [61] for a discussion of scalability issues in GrACE and [78] for CHOMBO.

Overture [65] had its beginnings in simulations using overset meshes i.e. a gridding scheme, generally applied to complex geometries, where the domain is discretized using "patches" which could employ meshing schemes that were best suited to the geometry at hand. Thus it was possible, for example, to embed a small circular patch, employing a cylindrical mesh, within a larger mesh discretized in a Cartesian manner. It was used to develop fourth-order-in-space discretizations [39], as well as in simulations of high-speed reactive flows [40]. It has been extended to adaptive mesh refinement [41], multi-material flows [8] and has recently been parallelized and used for 3D calculations [42]. A full listing of Overture-related publications, including investigation of detonations etc. can be found on the Overture homepage [65].

In the third category are frameworks that primarily seek to assist software developers implement a certain design pattern. CCA defines one such design pat-

tern; UINTAH [77] and CCAFFEINE [14, 1] are frameworks that implement it. Thus components designed to operate in the UINTAH and CCAFFEINE frameworks comply with the CCA architecture and bestow its advantages — modular design, reduction of software complexity, plug-and-play experimentation and multiple language interoperability — on simulation codes that follow the architecture. The framework is not required to provide any numerical or simulation capability (which renders the framework usable in diverse scientific applications) and indeed CCAFFEINE does not; however, UINTAH provides its users with a mesh and the associated data structures for describing fields on a discretized domain. A detailed discussion of CCA follows in Sect. 3.

3 The Common Component Architecture

The Common Component Architecture (CCA) specification defines a software standard allowing plug-and-play composition of scientific applications. Being component-based, it is necessarily object-oriented. Its development was driven by the benefits of modularization. The competitive advantages of modularization were recognized by the commercial establishment in the mid-1990s, which fashioned a solution in the form of component architecture; CORBA [22], Visual Basic [85] and Java Beans [33] are some industry-standard *component architectures*. These architectures employ certain subsets of object-oriented software design principles to realize modularity and interoperability of modules under a large variety of conditions. However, commercial component standards are ill-suited for scientific computing [1], the primary drawbacks being the lack of support for parallel computing. Starting from the concepts common to most component standards, the CCA retains most of the relevant characteristics while simultaneously allowing (but not stipulating) parallel computing in an SPMD fashion. CCAFFEINE [1] and UINTAH [77] are CCA-compliant frameworks that support parallel computing, while XCAT [87] does not.

Within CCA one considers *components*, modules (or objects) that implement a particular scientific or algorithmic functionality, and a *framework* e.g. CCAFFEINE or UINTAH, that "wires" components together into a functioning simulation code. Components implement interfaces (*Ports* in CCA parlance) through which the components provide their functionality; these ports are designed by the programmer implementing the component. Components are peers, i.e., they do not inherit implementation from other components, and are capable of being used independently. Since components implement interfaces, they can be developed without a tight coupling to a software development team. CCA stipulates that all components implement a particular interface (called the *Component* interface) through which individual components can interact with the framework (specifically, using the *Services* interface). Typically, this involves registering the Ports (interfaces/functionalities) that they provide (these interfaces are called the *ProvidesPorts* of the component) as well as the Ports that they use (i.e., the *UsesPorts* of the component). Driven by a user script, the framework matches the *ProvidesPort* and *UsesPort* registrations.

The individual components then fetch the matched-up Ports and use them by making calls on the methods on those interfaces The Port interface is a caller-callee boundary, not a data-flow abstraction.

3.1 Features of the Common Component Architecture

The initial development of the CCA specification took place using a compiler-independent, simple subset of C++ syntax (pure virtual classes) as the interface definition language. Subsequent CCA development (version 0.6 and beyond) use the Scientific Interface Definition Language (SIDL) and its compiler Babel [23] to enable language interoperability i.e. components written in the Babel-supported languages (C, FORTRAN, Python, or Java) can call each other transparently, without manually performing the data translations and language-bindings that such a mixture of programming languages would entail. When using Babel, the interface construct of the SIDL language replaces the pure virtual abstract class of C++ (of the initial design). In addition to language translation, SIDL/Babel provides modern software engineering functionality in each of the languages it supports e.g., passing and accessing multidimensional array data and C structs as references to preserve performance, support for exceptions, object-oriented design, tunably enforced programming-by-contract conditions when entering and leaving method calls etc. The C++ specification continues to exist and provide interfaces equivalent to those of the SIDL definition.

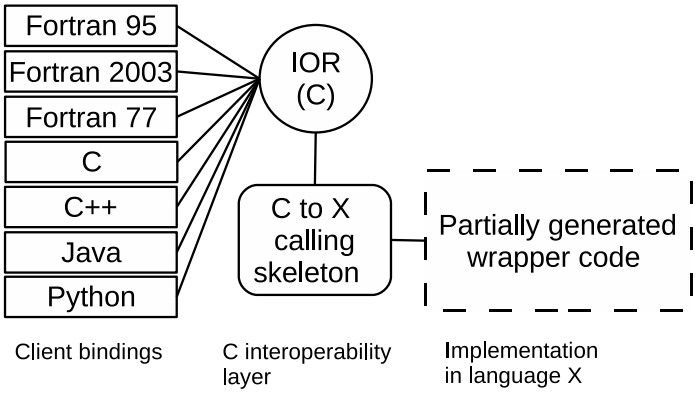


Fig. 1 Mechanics of any-to-any language calling with Babel. Client binding and interoperability layers (solid shapes) are entirely generated code. The implementation is partially generated in the implementor's choice of programming language, X, chosen from among the supported languages. The implementor fills in method bodies and private data structure definitions that are inaccessible from the clients. Lines indicate function calls.

Language interoperability: SIDL/Babel enables language-interoperability by interposing an Intermediate Object Representation (IOR) between components, with an independent client-side binding to each language. This is illustrated in Fig. 1.

In order that a component, written in language X, may be called from a host of other languages (in Fig. 1, these are various dialects of FORTRAN, C/C++, Java and Python), the programmer first expresses/writes the interface i.e., function calls, and their arguments (which may use the Babel-supported data-types like multi-dimensional arrays, exceptions etc. listed above) in a language-neutral manner using SIDL. The SIDL interface is then "compiled" using the Babel compiler to generate equivalent interfaces in C/C++, FORTRAN etc, collectively referred to as "client bindings" in Fig. 1 (left). It also generates a wrapper, in language X, (shown in dashed lines on the right side of Fig. 1) in which one implements one's components. The wrapper can also include calls to an external library, and in fact, provides a very simple route to making a library directly callable from a multitude of programming languages. The most significant, and intricate, piece in the cross-language orchestration is the IOR, shown in the middle of Fig. 1 ("C interoperability layer"). This IOR, essentially a C struct containing a function table and data pointers, maps function calls in the various "client" languages (left side of Fig. 1) into the equivalent functions/methods on the "server" side (the dashed structure on the right side of Fig. 1). The need for translation of data (e.g from FORTRAN multi-dimensional arrays on the "client" side to perhaps C on the "server"/implementation side), the throwing of exceptions and other functionalities that might be needed to enable cross-language operation are detected by Babel when "compiling" the SIDL interface, and is encoded into the IOR. There are a substantial number (as seen in Fig. 1) of code objects which are automatically derived from SIDL by Babel and then compiled. Most of the rote work needed to build and deploy SIDL-enabled applications has been automated with the Bocca tool [2], enabling entire component application structures to be prototyped within minutes.

Parallel computing: Since the use of parallel computing is critical to performance, the CCA standard is carefully crafted to avoid forcing the choice of any particular parallel programming model on the user. Component developers may use any communication model. A survey of the impact of using CCA on application performance is provided in Section 6 of [14]. The typical scientific application has a single-program-multiple-data structure (SPMD). This programming model is well supported by the CCAFFEINE framework [1] and default driver, which supports the component-based programming model analog, the single-component-multiple-data (SCMD) model. In the SCMD model, all components have a representative instance on all processes. For a single component, we call the group of parallel instances a cohort. Message passing, by whatever means, is restricted to exchanges within a cohort, as one component implementation cannot make any assumptions about the communication internals of a different component implementation. CCAFFEINE has also been used as a library to compose MCMD (multi-component-multiple-data) applications [46].

4 Computational Facility for Reacting Flow Science

The Computational Facility for Reacting Flow Science (CFRFS) is a toolkit for conducting high-fidelity simulations of laboratory-scaled flames. The toolkit adheres to the CCA specification (using C++, not SIDL/Babel as its interface language of choice) and is typically run using the CCAFFEINE framework. The CFRFS toolkit implements a set of novel numerical techniques, most of which were developed as the toolkit was being constructed. In this section we describe the toolkit and its numerical structure and discuss why a component architecture was necessary for its implementation. We conclude with an analysis, using data drawn from the toolkit, to determine to what degree the original aims of the software design have been realized.

4.1 Numerical Methods and Capabilities

The CFRFS toolkit is used to perform computations of flames using detailed chemical mechanisms. It solves the low Mach number approximation of the Navier-Stokes (NS) equations [86], augmented with evolution equations (convective-diffusive-reactive systems) for each of the chemical species in the chemical mechanism. It employs fourth-order finite-difference schemes for spatial discretizations within the context of AMR meshes [13]. It adopts an operator-split construction to enable the use of efficient integrators when time-integrating different physical processes in the system being solved. It uses extended-stability Runge-Kutta-Chebyshev (RKC) time-integration [76] for time-advancing the diffusive transport terms and an adaptive, backward difference stiff integrator for the chemical source terms. The numerical details of the high-order (spatial) methods can be found in [73] and those of RKC on block-structured adaptive meshes in [53]. The divergence constraint in the low Mach NS equations necessitates a projection scheme, which gives rise to a non-constant coefficient Poisson equation; this is solved using the conjugate-gradient method with a multigrid solver (employing high-order spatial stencils) as a preconditioner.

The CFRFS Toolkit, then, is an integration of many advanced numerical techniques. Many of them, e.g., RKC, had been demonstrated on uniform meshes [64] but had to be augmented to enable the solution of partial differential equations (PDEs) on AMR meshes. For example the RKC schemes had to be modified to preserve its order of accuracy when used with *time refinement* [53] on AMR meshes; further tests were required to determine various "free" parameters (in the RKC scheme) when time-advancing a convective-diffusive system [72]. The high-order spatial discretizations required appropriate interpolation schemes, and in certain cases, needed the solution to be filtered, to remove high-wavenumber content and prevent Gibbs phenomenon [73]; the correct pairings of discretization, interpolation and filter order were determined as a part of the implementation of the toolkit. The projection scheme adapts a fourth-order finite-volume formulation [44] for use in

the context of a finite-difference approach [74]. Thus the construction of the high-order AMR simulation capability, as implemented in the CFRFS Toolkit entailed a significant amount of development of advanced numerical methods.

The CFRFS Toolkit makes copious use of external software. The adaptive meshing and load-balancing is currently provided by the GrACE package [38]; coupling to CHOMBO [17] is in progress. The stiff integration capability is provided by CVODE [19], while the elliptic solvers in Hypre [34] are used for the pressure solve. Legacy codes are used to provide implementations of various constitutive models (transport coefficients, gas-phase reaction and thermodynamics models etc). Being able to leverage existing, validated software (e.g., legacy codes) has saved much implementation effort, while numerical libraries (e.g. Hypre, CHOMBO) allow the CFRFS toolkit to take advantage of optimized and specialized capabilities in a facile manner.

4.2 The Need for Componentization

The goal of the CRFRS development effort was to develop a flexible, reusable toolkit. At the very outset it was expected that many of the its advanced features would be contributed piecemeal by experts or incorporated using legacy software and the necessity of an extremely modular design was recognized very early.

The componentization in the CFRFS toolkit follows strictly along functional lines. Each component implements a physical model, a numerical scheme, or a computational capability like a "data object" that stores and manages domain-decomposed fields (e.g., a temperature field) on multiple levels on an AMR mesh. The functionality is expressed in the Port/interface design; components implement the functionality. As there are may different ways to provide a functionality e.g., one may time-integrate using many different algorithms or calculate transport properties using diverse models, a single Port may find disparate implementations. Each component is compiled into a shared library (also known as a dynamically loadable library); a simulation "code" is composed by loading a number of them into the CCA framework and "wiring them together". Fig 2 shows a wiring diagram, assembling approximately 40 components into a low-Mach number flame simulation, whose results are discussed in Sec. 5.1. The components in the wiring diagram implement flow models (fourth-order discretizations for convective and diffusive fluxes, detailed chemical models etc), numerical schemes (the pressure solution, sixth-order interpolation schemes), the AMR mesh and the associated data object and miscellaneous components for I/O etc. The components can be approximately collated into 3 sub-assemblies, responsible for scalar transport, momentum transport (including the projection required for solving the low-Mach number approximation of the Navier-Stokes equation) and for advancing reactive terms. The components are dynamically loadable, and so, the "code" is composed at runtime. The components and wiring connectivity are specified in an input file to the framework; components can be exchanged simply by changing a single line in this input file.

The aim of componentization was the taming of complexity. One measure of complexity is the pattern in which different components might use each other. A good design would exhibit modularity, where connectivity between components is sparse and connections are arranged in some regular manner e.g., if components are collected/connected into sub-assemblies, which are hierarchically composed into the simulation code. Fig. 2 shows 3 separate sub-assemblies consisting of components that address the transport of species, chemical reactions and the momentum solve (including the pressure solution). The size of each component is a second measure of complexity; smaller components are easier to understand and maintain. In Fig. 3 (left), we plot a histogram of the size (lines of code) in a component; it is clear most components are small, less than 1000 lines of code. Since components implement Ports, this strongly suggests that individual Ports do not embody much complexity either i.e., they have few methods that need to be implemented. This is shown in Fig. 3 (right) where we plot a histogram of the number of methods in each port. Most of the ports have 10 or fewer functions/methods. Figs 4 (left) plots the histogram of the number of ProvidesPorts i.e, the number of Ports a given component implements. It is clear that the bulk of the components implement less than five ports each, which explains their small size; recall that most ports have fewer than 10 methods. Another measure of a component's complexity or importance is the number of UsesPorts it has. Components that link disparate sub-assemblies together tend to have many UsesPorts distributed among the sub-assemblies. In Fig. 4 (right) we plot the number of UsesPorts per component, which is proportional to the number of other components a given component requires to perform its functions. We see that almost all components have less than 10 UsesPorts. Since components provide, on an average, less than 5 ports each (see Fig. 4, left), a component is connected to approximately 2-3 other components, leading to sparse connectivity between components. These statistics show that a relatively sparse specification of functionalities and interconnections may suffice for the construction of quite complex scientific software.

5 Computational Investigations Using CCA

In the previous section we described how CCA was used to architect and implement the CFRFS toolkit whose design philosophy stressed small, simple components, sparse connectivity between components and their hierarchical composition, via sub-assemblies, into functioning simulation code. In this section, we demonstrate two different ways in which the components of the the CFRFS toolkit are used.

The CFRFS toolkit consists of two sets of components. The first set, by far the bigger one, consists of components that address the numerical issues surrounding the use of fourth- (and higher) order spatially accurate methods on block-structured AMR meshes. These allow efficient resolution of fine flame structures without the necessity of overwhelming computational resources. The second set of components

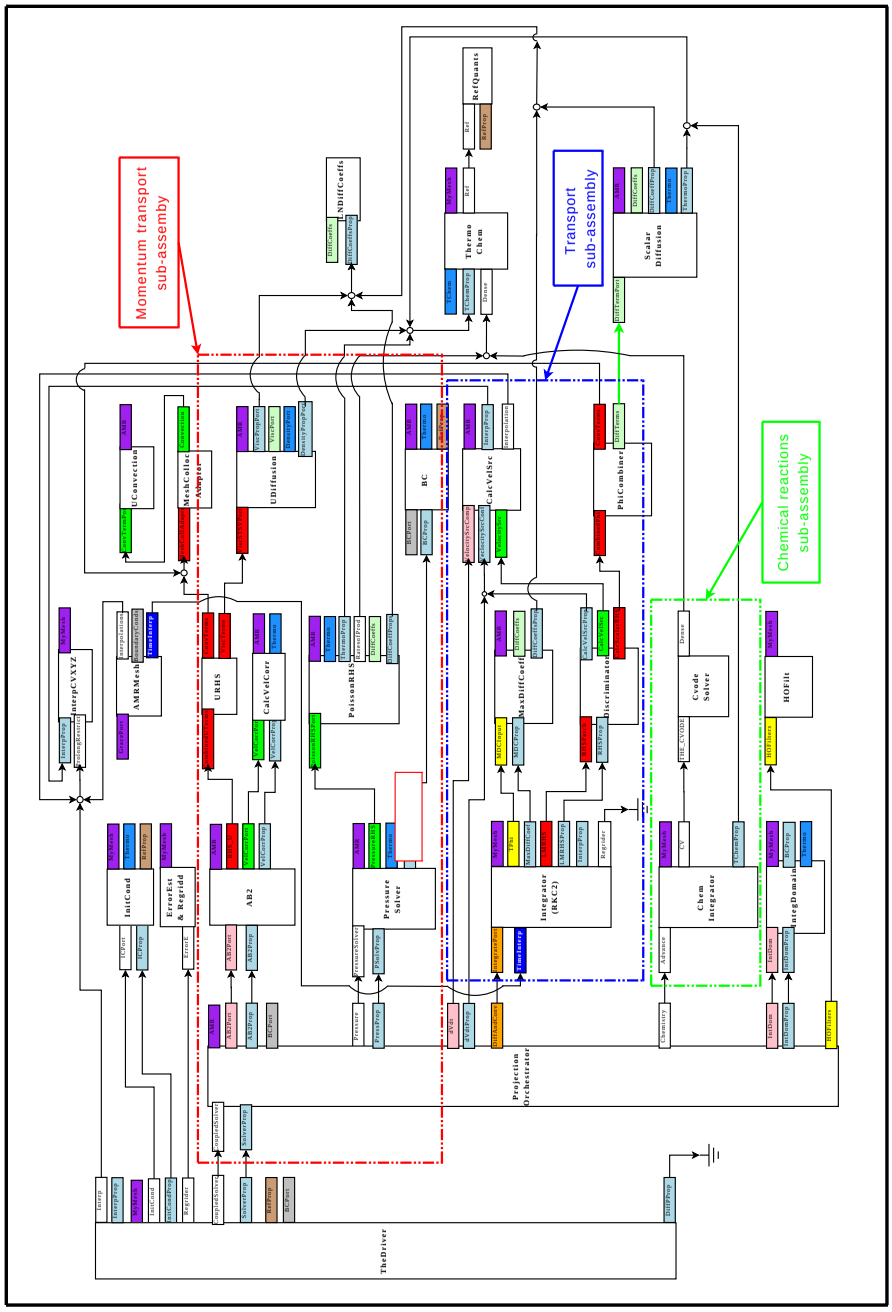


Fig. 2 Wiring diagram of the approximately 40 components that are used for simulating reactive flows in the CFRFS toolkit. The components are shown in white; different ports are colored. The ProvidesPorts appear on the left of a component; the UsesPorts on the right. The components are assembled, roughly, into 3 sub-assemblies, responsible for scalar transport (blue dash-dotted box), momentum transport (red dash-dotted box) and for modeling reactive processes (green dash-dotted box). The components can be strictly numerical (e.g., interpolation, pressure solver), models (detailed chemistry, diffusion coefficients) or "computer science" e.g., the AMR Mesh and the data object.

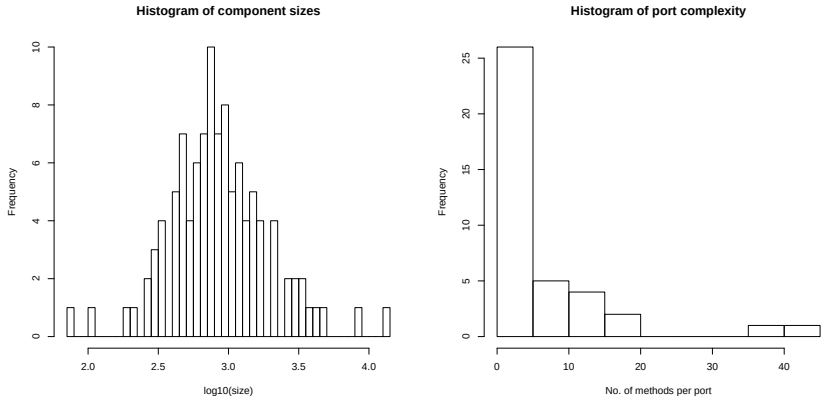


Fig. 3 Left: Histograms of the logarithm (to base 10) of component sizes, measured as the total number of lines, including comments and blank lines. The histogram clearly shows that half the components are less than 1000 lines long, and almost all are less than about 3000 lines. Components, therefore, are generally quite small. Right: Histogram of the number of functions/methods per port. We see that most ports have less than 10 methods each. There are a couple of ports, related to the mesh and the data object that have approximately 40 methods each. These statistics were extracted from a population of 100 components

addresses the identification of low-dimensional manifolds in the chemical dynamics, so that chemical source terms may be tabulated and thus evaluated inexpensively within the context of spatially resolved flame simulations. This is done using Computational Singular Perturbation [51]. Many components, for example, those modeling chemical reactions, thermodynamics and constitutive models find use in both the efforts. The final goal is to replace/augment the reactive subsystem, consisting of a stiff-integrator and the chemical source terms (as described in Sec. 4), with an inexpensive tabulation scheme that would allow the toolkit to be used with large (and stiff) chemical mechanisms typically associated with higher hydrocarbons.

5.1 Fourth-order Combustion Simulations with Adaptive Mesh Refinement

Chemically reacting flow systems based on hydrocarbon fuels typically exhibit a large range of characteristic spatial and temporal scales. The complexity of kinetic models, even for simple hydrocarbon fuels, compounds this problem, making multidimensional numerical simulations difficult. This is true even for laboratory scale configurations.

These difficulties are commonly addressed in a variety of ways. For low speed flows, one may adopt a low Mach number approximation [58] for the momentum

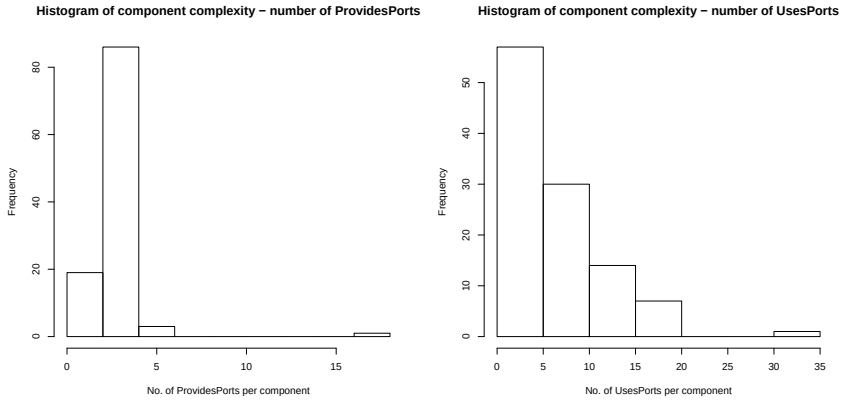


Fig. 4 Left: Histogram of the number of ProvidesPorts implements by the components in the CFRFS toolkit. We see that most of the components have 4 or fewer ProvidesPorts i.e., they implement very few ports. The distribution of the number of methods in each port is plotted in Fig. 3 (right). Right: Histogram of the number of UsesPorts a component uses. This is *one* measure of the complexity of the algorithm/functionality a component implements. Components that link subassemblies of components also tend to have many UsesPorts. We see that most components have less than 10 UsesPorts.

transport. This approximation assumes that acoustic waves travel at infinite speed, a justifiable assumption in many low-speed flows. One can also exploit the structure of the governing equations and adopt an operator-split mechanism, performing the transport and reactive time-advancement via specialized integrators [64]. In problems where fine structures exist only in a small fraction of the domain e.g., in laminar jet flames, one may employ AMR [13] to concentrate resolution only where needed [12, 71, 24, 6], while maintaining a coarse mesh resolution elsewhere.

The CFRFS toolkit implements a numerical model that can efficiently simulate flames with detailed chemical mechanisms. The use of AMR is not without its challenges, beyond just programming complexity. In order to reduce the number of grid points and the number of refinement levels in the computational mesh hierarchy we employ high-order stencils to discretize the governing equations and to interpolate between the computational blocks on adjacent mesh levels. A projection scheme is employed for the momentum transport. Since mesh adaptivity is driven by the narrow flame structure rather than the velocity field, we solve the momentum transport on the lowest level mesh in the AMR mesh hierarchy i.e., on a uniform mesh. This further enhances the efficiency of the model since the elliptic solver required by the pressure equation is more efficient on a uniform mesh, compared to a multilevel one [59]. The numerical approach and results obtained for canonical configurations are presented below.

5.1.1 Formulation

In the low-Mach number limit, the continuity, momentum and scalar transport equations for a chemically reacting flow system are written in compact form as

$$\nabla \cdot \mathbf{v} = -\frac{1}{\rho}\frac{D\rho}{Dt} \tag{1a}$$

$$\frac{\partial \mathbf{v}}{\partial t} = -\frac{1}{\rho}\nabla p + C_U + D_U \tag{1b}$$

$$\frac{\partial T}{\partial t} = C_T + D_T + S_T \tag{1c}$$

$$\frac{\partial Y_k}{\partial t} = C_{Y_k} + D_{Y_k} + S_{Y_k} \qquad k = 1, 2, \ldots, N_s. \tag{1d}$$

Here $\mathbf{v}$ is the velocity vector, ρ the density, T the temperature, Y_k the mass fraction of species k, p is the hydrodynamic pressure, and N_s is the number of chemical species. The $\frac{D}{Dt}$ operator in the continuity equation represents the material derivative, $\frac{D}{Dt} = \frac{\partial}{\partial t} + \mathbf{v} \cdot \nabla$. The system of equations is closed with the equation of state for an ideal gas. The thermodynamic pressure spatially uniform and is constant in time for an open domain in the low-Mach number limit. NASA polynomials are used to compute thermodynamic properties [60]. The transport properties are based on a mixture-averaged formulation and are evaluated using the DRFM package [68].

The equation of state is used to derive an expression for the right hand side of the continuity equation (Eq. 1a)

$$\frac{DP_0}{Dt} = 0 \rightarrow \frac{1}{\rho}\frac{D\rho}{Dt} = -\frac{1}{T}\left(D_T + S_T\right) - \sum_{k=1}^{N_s} \frac{\bar{W}}{W_k}\left(D_{Y_k} + S_{Y_k}\right) \tag{2}$$

5.1.2 Implementation in the CCA Framework

The numerical integration of the system of equations (1a-1d) is performed in three stages. In the first stage, a projection scheme is used to advance the velocity field based on the equations (1a,1b). Figure 5 shows the main CCA components involved in the momentum solver. The *Momentum Driver* component advances the velocity field to an intermediate value based on convection RHS_{conv} and diffusion RHS_{diff} contributions to the right-hand-side (RHS) term Vel_{rhs} of the momentum equation (1b). This is followed by an elliptic solve in the *Pressure Solver* component for the dynamic pressure p. The RHS values for the elliptic pressure equation are computed in $Pressure_{RHS}$. Transport and thermodynamic properties are provided by *Transport Properties* and *Thermo & Chemistry* components, respectively. The gradient of the pressure field is used to correct (Vel_{corr}) the intermediate velocities above to obtain a field that satisfies both the continuity and momentum equations (1a,1b). The components shown at the top of Fig. 5 (AMR Mesh, Boundary Conditions, Interpolations and Derivatives) are generic components that handle the adaptive mesh

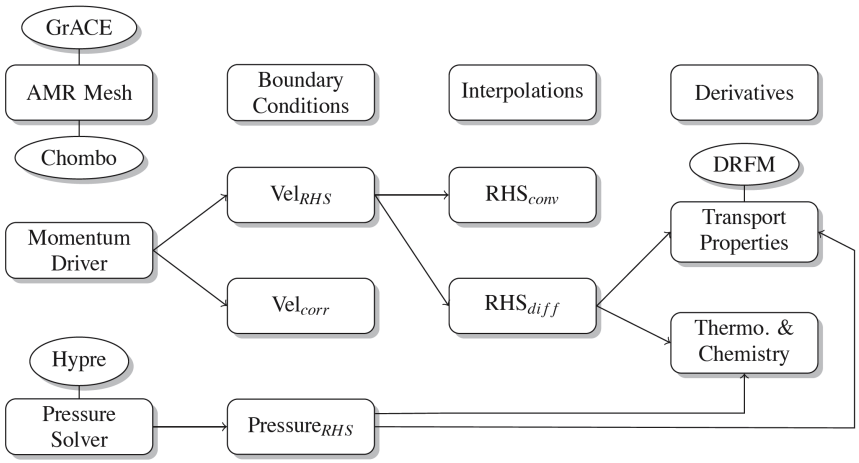


Fig. 5 Schematic of the momentum solver components in CFRFS.

refinement library, boundary conditions, interpolations, and derivatives. External libraries are shown in ellipses.

In the second stage, sketched in Fig. 6, temperature and species mass fractions are advanced using an operator split approach that separates the convection, C_T, C_{Y_k}, and diffusion, D_T, D_{Y_k}, contributions from the ones due to the chemical source terms, R_T, R_{Y_k}, in Eq. (1c,1d). Symmetric Strang splitting is employed, beginning with the chemical source term contribution for half the time step, followed by the contributions from convection and diffusion terms for a full time step, and concluded by the remaining contribution from the reaction term for half the time step. The scalar advance due to the chemical source term is handled by *Chemistry Integrator*. The convection ($Scalar_{conv}$) and diffusion ($Scalar_{conv}$) contributions are combined by $Scalar_{RHS}$ component and provided to *RKC2 Integrator* which uses a Runge-Kutta-Chebyshev (RKC) algorithm [84] for time advancement. A *Switchboard* component is used to ensure that velocities are available at intermediate times during the multi-stage RKC integration.

The third stage repeats the projection algorithm from the first stage using the updated scalar fields from second stage. The overall algorithm is 4^{th}-order accurate in space and 2^{nd}-order in time.

Adaptive mesh refinement We employ an AMR approach where the computational domain is split into rectangular blocks. The advancement in time of the AMR solution is based on Berger-Colella time refinement [13, 53]. Figure 7 shows a schematic of this recursive time integration algorithm. Consider the solutions on levels L and $L+1$ at time t_n. Level L is first advanced to $t_n + \Delta t$, then the solution on $L+1$ is advanced in two half steps, $\Delta t/2$ to ensure numerical stability on the finer grid. During time advancement on $L+1$, boundary conditions are computed by interpolation using the solution on L. At $t_n + \Delta t$ the solution on $L+1$ is interpolated down to the

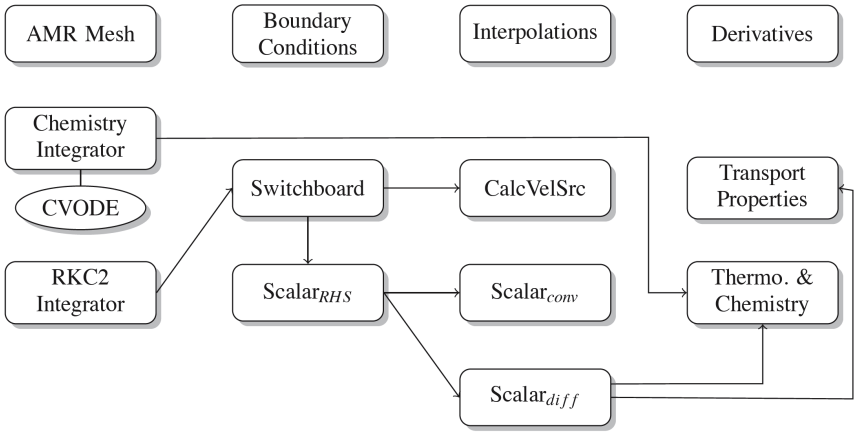


Fig. 6 Schematic of the scalar solver components in CFRFS.

corresponding regions on level L. In order to preserve the 4^{th}-order spatial convergence of the numerical scheme, the interpolations between adjacent grid levels use 6^{th}-order spatial stencils [73].

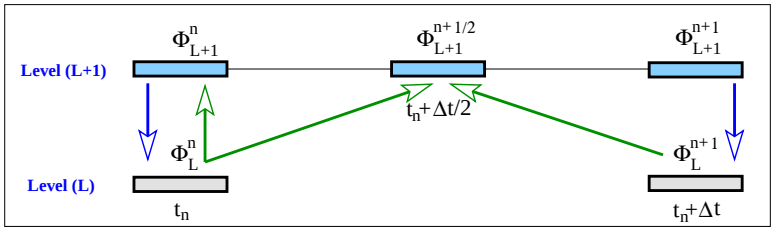


Fig. 7 Schematic of the time refinement in the context of AMR.

5.1.3 Application to Flame-Vortex Interaction

A canonical vortex-flame configuration [64] was chosen to explore the performance of the numerical construction. The computational domain is $1.5\text{cm} \times 0.75\text{cm}$. The velocity field corresponding to a periodic row of counter-rotating Lamb-Oseen vortices is superimposed over the premixed 1D flame solution discussed above. A relatively coarse mesh was used for the base mesh, with a cell size of 50μm in each direction. Additional, finer, mesh levels were added in the flame region during the simulation.

A one-step, irreversible Arrhenius global reaction model is used in addition to a C1 kinetic model to study the vortex-flame interaction. Figure 8 shows freeze frames of the vorticity and heat release rate fields. The vortex pair is initially located 2mm upstream of the flame and propagates with approximately 10m/s towards it. As the vortex pair impinges into the flame, the flame intensity decreases on the centerline for the C1 model while the one-step solution shows little change in the interaction region. Similarly, at locations off-centerline the flame intensity for the C1 model decreases significantly as it stretched and rolled around the vortex pair. The last frames show a significantly contorted flame, and the relative increase in the overall burning rate is about about 50% more for the one-step reaction simulation compared to the simulation using the C1 model .

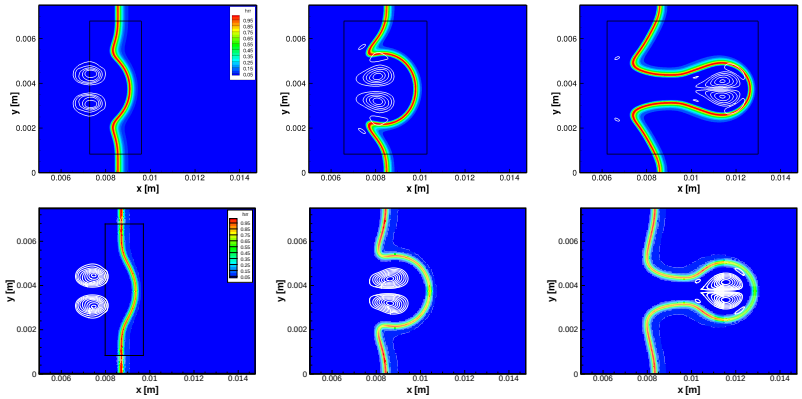


Fig. 8 Vorticity (white solid contours) and normalized heat release rate (hrr, shaded contours) for simulations using the one-step reaction model (upper row) and a C1 kinetic model consisting of 16 species and 46 reactions (lower row).

5.2 Computational Singular Perturbation and Tabulation

In the previous section, we described our experience with simulating flames using one-step and C1 chemical mechanisms. The primary challenge in going from simple one-step chemistry to a C1 mechanism was the steep increase in the stiffness of the dynamical system composed of the reactive processes; its main effect was to reduce the size of the time-step one could take without unacceptable splitting errors. Matters are further compounded when one considers C2 (or even more detailed) chemical mechanisms. This stiffness of detailed chemical mechanisms is due to the wide range of time scales that they model. It leads to considerable difficulties when time-advancing them in an efficient manner. Chemical model simplification and reduction strategies typically target these challenges by reducing the number of reac-

tions and/or species in the model, with associated reduction in model complexity. When done properly, this strategy also reduces the system stiffness. Alternatively, the Computational Singular Perturbation (CSP)-based time integration construction of [83] uses CSP analysis to project out the fast time scales from the detailed chemical source term, thereby rendering the equations non-stiff. The promise of this approach is that explicit time integrators can be used for large-time step integration of the resulting non-stiff source terms, and could potentially eliminate the need for operator-split time integration of reaction-diffusion source terms.

The key challenge with this time integration approach, however, is the large computational cost of solving for the requisite CSP information and the resulting projection matrices. An approach to mitigate this computational cost is tabulation. By adaptively storing and reusing the CSP information, the significant CSP overhead can be drastically reduced (by amortization), leading to an efficient overall implementation. Such a tabulation strategy has been explored for elementary model problems [51, 49]. This section describes the implementation of a CSP tabulation approach, relying on *k*d-trees [7] to efficiently store and retrieve CSP information along manifolds in the chemical configuration space.

In the following, the CSP time integration and tabulation approach is formulated, followed by a discussion of its implementation in the CCA framework for reacting flow simulation. Next, the approach is illustrated on the simulation of H_2 – air ignition.

5.2.1 Formulation and Implementation

Consider the chemical system described by $\mathrm{d}\boldsymbol{y}/\mathrm{d}t = \boldsymbol{g}(\boldsymbol{y})$, where $\boldsymbol{y} \in \mathbb{R}^N$, and $\boldsymbol{g}(\boldsymbol{y})$ is the chemical source term. The CSP basis vectors $\{\boldsymbol{a}_k\}_{k=1}^N$ and covectors $\{\boldsymbol{b}^k\}_{k=1}^N$, all in $\mathbb{R}^N$, enable the decoupling of the fast and slow processes, and the identification of low dimensional slow invariant manifolds (SIMs) [48]. Thus, we have

$$\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}t} = \boldsymbol{g} = \boldsymbol{g}_{\mathrm{fast}} + \boldsymbol{g}_{\mathrm{slow}} = \boldsymbol{a}_1 f^1 + \boldsymbol{a}_2 f^2 + \cdots + \boldsymbol{a}_N f^N \tag{3}$$

where $f^i = \boldsymbol{b}^i \cdot \boldsymbol{g}$, for $i = 1, 2, \ldots, N$. In this equation, $\boldsymbol{g}_{\mathrm{fast}}$ corresponds to the modes with fast transients, which are rapidly exhausted. After relaxation of fast transients, with M modes exhausted, $\boldsymbol{g}_{\mathrm{fast}} = \sum_{r=1}^{M} \boldsymbol{a}_r f^r \approx 0$ and $\boldsymbol{g}_{\mathrm{slow}} = \sum_{s=M+1}^{N} \boldsymbol{a}_s f^s = (I - \sum_{r=1}^{M} \boldsymbol{a}_r \boldsymbol{b}^r)\boldsymbol{g} = \boldsymbol{P}\boldsymbol{g}$. In practice, the number of exhausted modes is determined as the maximum M for which $\tau_{M+1} \sum_{r=1}^{M} \boldsymbol{a}_r f^r$ is less than a user-specified threshold, where τ_{M+1} is the time scale corresponding to the $(M+1)^{st}$ mode.

The CSP integrator [83] proceeds in each time step by first integrating the slow dynamics of the system, followed by a homogeneous correction (HC) to correct for the fast time scales:

$$\tilde{\boldsymbol{y}}(t+\Delta t) = \boldsymbol{y}(t) + \int_t^{t+\Delta t} \boldsymbol{P}\boldsymbol{g}\,\mathrm{d}t' \tag{4}$$

$$\boldsymbol{y}(t+\Delta t) = \tilde{\boldsymbol{y}}(t+\Delta t) - \sum_{m,n=1}^{M} \boldsymbol{a}_m \tau_n^m|_t \hat{f}^n \tag{5}$$

$$\hat{f}^n = \boldsymbol{b}^n \cdot \boldsymbol{g}[\tilde{\boldsymbol{y}}(t+\Delta t)] \tag{6}$$

where τ_n^m is the inverse of λ_n^m, given by

$$\lambda_n^m = \left(\frac{\mathrm{d}\boldsymbol{b}^m}{\mathrm{d}t} + \boldsymbol{b}^m \boldsymbol{J}\right)\boldsymbol{a}_n \tag{7}$$

and $\boldsymbol{J}$ is the Jacobian of $\boldsymbol{g}$. The matrix τ_n^m is diagonal with entries the time scales $\{\tau_k\}_{k=1}^N$ when the CSP basis vectors are chosen to be the eigenvectors of $\boldsymbol{J}$ and the curvature of the SIM is neglected, *i.e.* $\mathrm{d}\boldsymbol{b}^m/\mathrm{d}t = 0$.

The procedure outlined above separates the fast, exhausted modes from the slow modes that drive the evolution of the system along the SIMs. As discussed in [47], CSP also identifies the species that are associated with these fast modes as *CSP radicals*. (These are the species whose concentration can be determined from the algebraic equations resulting from setting $f^i = 0, \quad i = 1,\ldots,M$.) Accordingly, the species space can be separated into the CSP radicals and non-CSP radicals.

To improve the efficiency of the CSP integrator, a tabulation approach has been developed to enable reuse of the essential CSP quantities: the M fast CSP vectors and covectors, as well as the $M+1$ fastest time scales, which are sufficient to assemble the slow-manifold projector $\boldsymbol{P}$ needed for the HC and CSP integration, and to select the time step along the slow manifold. As the CSP vectors, covectors and time scales can be modeled as functions of the non-CSP radical species only, it is sufficient to tabulate these quantities in an $N-M$ dimensional table, rather than having to cover the full N-dimensional state space.

In the work presented here, a table with manifold conditions is constructed offline, by performing full CSP analysis on a number of *design points* in state space. We first randomly sample a set of initial conditions over a range of initial temperatures, equivalence ratios and N_2 dilution factors (extra mole of N_2 per mole of air) and integrate them forward, with CVODE [19], using detailed reaction kinetics. A set of design points is constructed from the system states encountered during those simulations. For each of these design points, a CSP analysis is performed to identify associated SIMs. If a design point has exhausted modes, then successive HCs are applied to project that design point onto the corresponding SIM. Each SIM is characterized by a unique value of M and the associated CSP radicals.

For each identified SIM, the tabulation of the associated CSP information relies on a nonparametric regression approach. For this purpose, the CSP information is stored in *k*d-tree data structures over the the $N-M$ dimensional space of the non-CSP radical species. Note that, in order to give equal weight to all dimensions of the state vector in the computation of distance measures, all coordinates of the manifold points are first rescaled and shifted to range between 0 and 1 before being stored in the *k*d-trees. During time integration, the manifold that best corresponds to the

current condition in the chemical configuration space is determined by finding the nearest neighbor, as measured by the Euclidean distance measure, in all of the manifolds in the table. If the manifold point that is closest to the current condition over all manifolds is within a maximum allowable distance d, then the associated manifold is assumed to be the one that is currently being followed by the system. The CSP information at the current condition is then approximated with the corresponding values at the nearest neighbor point in the table, which amounts to a 0^{th}-order interpolation. Higher order interpolations, relying on interpolation between nearest neighbors or on polynomial response surfaces [50, 81], are the subject of ongoing work. In case none of the nearest neighbors in the tabulated manifolds are within the maximum allowable distance, then a full CSP analysis is performed on the current condition instead.

To implement the CSP integration approach, extensive use was made of existing components in the CCA framework. For example, the evaluation of the chemical kinetics source term and its Jacobian rely on the "AMR" set of components from CFRFS toolkit discussed in Sect. 4. Time integration relies on a CVODE component, part of the CFRFS toolkit. New components were developed to perform the CSP analysis as well as the table construction and interpolation for the tabulation approach. These components were joined together through the use of driver components that organize the overall algorithmic sequence of operations.

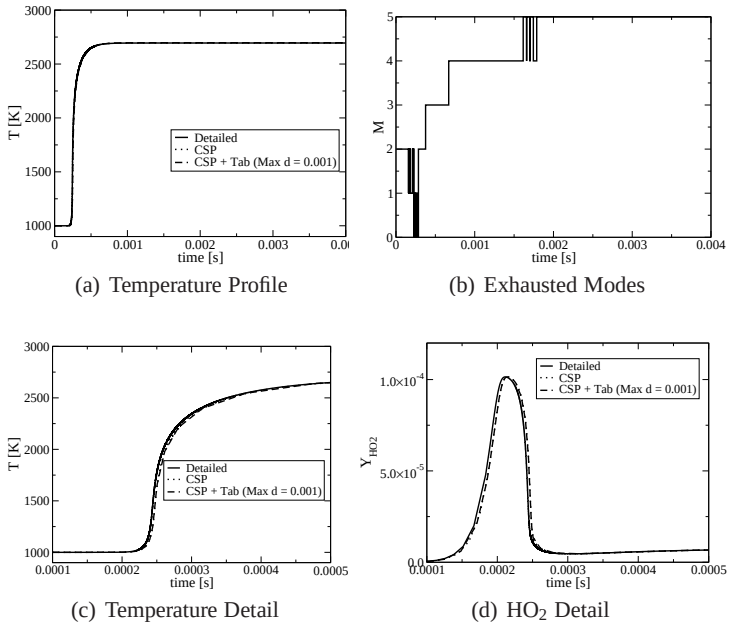
5.2.2 Application to H_2–air Ignition System

The CSP integration method outlined in the previous section was applied to the simulation of ignition of a stoichiometric homogeneous H_2–air mixture at a temperature of $T = 1000$ K. The system is modeled using a 9 species reaction mechanism, resulting in a total state space dimension of $N = 10$ (9 species + temperature) [88]. Fig. 9 compares the predicted temperature evolution obtained by integrating the detailed reaction kinetics (with the implicit solver CVODE), to the solution obtained with the CSP integrator (using the explicit fourth order Runge-Kutta (RK4) integration scheme), and with the CSP + tabulation approach .

For the tabulation approach a CSP table was constructed by sampling 100 initial conditions with Latin Hypercube Sampling over a range of equivalence ratios between 0.9 and 1, initial temperatures between 980 and 1020 K, and dilution factors between -0.005 and 0.005. From the design points extracted from these runs, close to 1 million states were identified on 9 different manifolds, with a number of exhausted modes ranging from 1 to 5.

The CSP integrated solution, both with and without tabulation is in good agreement with the full solution, except for a small difference in the ignition time delay, as is shown in detail in Fig. 9(c) and 9(d).

Note that, as the reaction progresses, the number of exhausted modes M, and the associated CSP radicals change according to the reaction dynamics. Fig. 9(b) indicates that the system initially has two exhausted modes, followed by a time window during ignition where all modes are active, after which M gradually increases



(a) Temperature Profile (b) Exhausted Modes

(c) Temperature Detail (d) HO_2 Detail

Fig. 9 a) Evolution of temperature in an igniting stoichiometric H_2–air system, simulated using the detailed reaction mechanism, the CSP solver, and the CSP solver with tabulation. b) Evolution of the number of exhausted modes, M, as obtained by CSP analysis. All approaches are in good agreement, except for minor differences in the ignition time delay, as shown in the close-up of the ignition zone for c) temperature and d) one of the trace species, HO_2. The initial conditions were: $T = 1000$ K, $Y_{H_2} = 0.0285$, $Y_{O_2} = 0.2264$ and $Y_{N_2} = 0.7451$.

up to five at late time, as more and more modes become inactive. Accordingly, as the number of exhausted modes increases, the tabulation approach becomes more efficient in terms of storage and lookup times as the CSP information for each manifold is tabulated in $N-M$ dimensional *k*d-trees. For example, for the H_2–air system studied here, tabulation in a 5-dimensional table is sufficient for the section(s) of the 10-dimensional state space where 5 modes are exhausted (see Fig. 9(b)).

In terms of efficiency of the table usage, Fig. 10 shows the number of successful table hits as a fraction of the total number of table lookups. As a table lookup is performed in every time step, this number indicates how efficient the tabulation approach is at avoiding full CSP analyzes by providing tabulated CSP information instead. For the current table and initial condition, the table lookup success rate increases from 25 % to about 65 % as the maximum allowed nearest-neighbor distance is increased from 0.001 to 0.03, while the accuracy of the integration does not noticeably change (not shown here). Other numerical experiments indicate that this table hit success rate and the accuracy of the tabulation assisted simulations also depends on the density of the table in state space. A quantitative relationship between the table density, the maximum allowed distance in the nearest neighbor

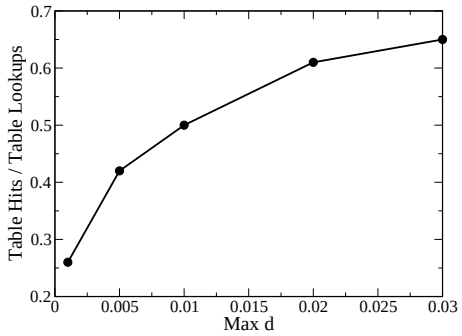


Fig. 10 The number of table hits, as a fraction of the total number of table lookups, increases as the maximum allowed distance to the nearest neighbor on a manifold is increased, resulting in more efficient usage of the tabulated data.

table lookup, and the accuracy of the CSP tabulation approach is the subject of ongoing work.

A comprehensive evaluation of the overall numerical performance of the CSP integration scheme with tabulation, as a function of table size, system and manifold dimensionality, degree of stiffness, and desired accuracy, is currently in progress. However, preliminary performance measurements show the CSP tabulation scheme to be competitive with direct CVODE integration for the cases studied in this paper.

To aummarize, CCA provides a flexible framework for the implementation of several components for the integration of stiff chemical kinetics. The modularity of the framework allows easy reuse of components that were developed elsewhere in the project for operations such as time integration, or the evaluation of source terms or Jacobian. This flexibility allows the rapid development of codes to test various integration approaches and easy switching between them for method comparison and validation.

6 Research Topics in Computational Frameworks

The previous sections have described and demonstrated how modularity, obtained as a result of adopting a component-based design, may be used to mitigate the effects of software complexity. They have also shown the sophistication of the scientific software that can be designed using a component-based approach. However, the experience with CFRFS, as well as other componentization efforts [45, 46], have revealed a number of difficulties, Some solutions have recently been crafted, which we discuss below.

The Learning Curve: A significant challenge in adopting CCA has been the learning curve associated with using SIDL/Babel. While this was not encountered when developing the CFRFS toolkit (which uses the original C++-interface approach, not SIDL/Babel), it was observed in other componentization efforts [45, 46].

The process of generating client- and server-side code, as described in Sec. 3.1, is prone to error if performed manually, but can be automated. An integrated development environment, called Bocca [2], has been developed (see [16] for Bocca tutorials) for this purpose. Given the interfaces that a component uses and provides, Bocca automatically invokes Babel, creates the client- and server-side auto-generated code and constructs a build system to compile the resulting component skeleton. It minimizes what a CCA developer has to learn, enabling him/her to focus on more productive tasks.

Reluctance to Abandoning Working Software: The process of componentization could be significantly simplified if one could automatically *derive* components out of a non-component codebase. A concept, called OnRamp [43], is being investigated by CCA researchers to enable such an automated derivation of components. OnRamp is driven by annotations which are inserted into the codebase (indicating interfaces, code-blocks that will reside in components etc.), from which is it possible, under certain restrictions, to automatically generate components and its associated build system. This preserves the original code and most of the software development practices that the programmer is familiar with, while bestowing the benefits of componentization on the software in question.

Components also confer benefits beyond the fundamental requirement of constraining software complexity. Since components are a black box regarding implementation but adhere to a specified convention for communicating with the outside world, they are ideal for automating computation at a high level; CCA has focused on performance improvements as the aim of automation. Below, we mention some of the recent advances in this arena.

Automatic Proxy Generation. The collection of performance characteristics on a method-call basis is a required, but laborious task in performance modeling. In a component environment, this can be achieved quite easily by exploiting the fact that components publish the interfaces that they use and provide. The collection of performance data can be done by interposing a proxy component between an interface provider component and an interface using component. The proxy component serves to trap calls between components and switch-on/terminate the collection of performance metrics (elapsed time, cache misses, page faults etc) by a performance measurement tool e.g. TAU [79]. Such "performance-measurement" proxies can be generated automatically [82]. They can be used to collect performance data and identify bottlenecks; they can also be used to *monitor* an executing simulation and *optimize* it during runtime. This is described below.

Computational Quality of Service: Since the framework has a holistic view of the entire application, and proxy components can monitor the performance of individual components, it becomes possible to manipulate their behavior, with a view to ensuring robustness, celerity of computation etc. This is commonly referred to as "Computational Quality of Service" [62]. The manipulation of components may be performed by changing parameters that a component may provide or by replacing entire components [56, 57]. See [62] for how this may be performed without modifying any components; a working example, using a shock-hydrodynamics simulations in CCAFFEINE, can be found in [56].

7 Conclusion

In this chapter, we have investigated how a component architecture may be used to design and implement scientific simulation software. The Common Component Architecture was chosen because of its ability to accommodate parallel computing. Unlike many computational frameworks, a component framework does not require one to "marry into" a prescribed set of data and code structures; in many cases, such "marriages" lead to one's dependence on the framework for the integration of external libraries and/or legacy software. The peer nature of components (whereby all components are independent) prevent such dependencies from arising. However, it is to be noted that a software architecture merely lays down a few software development principles; their judicious use is a matter of software design. The design, in turn, is dictated by where one starts from (i.e., whether one starts with a *tabula rasa*, which was our case, or whether one starts componentizing a legacy code) and what one wishes to achieve with the particular design. Any lack of clarity regarding the second aspect invariably leads to an unsatisfactory end.

In Sect. 4 we described the particular ends that wished to achieve with our component-based design, particularly maintainabiliy and reduction in software complexity; the statistics drawn from the 100 components in our toolkit provide some confidence that we have largely succeeded. In Sect. 5 we showed how the toolkit is used, including a few example of component reuse. Though not described here, some of the AMR components used in Sect. 5.1 have also been used to simulate problems in shock-hydrodynamics [52]. Thus there is some evidence to indicate that the plug-and-play promise of component software, widely realized in non-scientific software, may be replicated in our field too.

This chapter has drawn examples from the CFRFS effort, which emphasized small, manageable components designed without any constraints imposed legacy software. There are other efforts where legacy software has dictated both the aims and the course of componentization (see [45] and references within), and still others, usually involving the componentization of libraries, where users played a role (see Sect. 11 in [14]). Component-based design, and CCA in particular, is a versatile methodology for designing and developing maintainable software, and is most profitably used when one has a clear idea of *why* one wishes to use it. Unlike many frameworks developed to enable the *rapid prototying* of codes, its attractiveness lies in the long term.

Acknowledgements: The work documented in this chapter was funded by the Department of Energy, under its Scientific Discovery through Advanced Computing (SciDAC) program. Some of the computations were performed at the National Energy Research Supercomputing Center (NERSC) in Oakland, CA. The work was performed in Sandia National Laboratories, CA. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

References

[1] Allan BA, Armstrong RC, Wolfe AP, Ray J, Bernholdt DE, Kohl JA (2002) The CCA Core Specifications in a Distributed Memory SPMD Framework. Concurrency-Pract Ex 14:323–345, also at http://www.cca-forum.org/ccafe03a/index.html

[2] Allan BA, Norris B, Elwasif WR, Armstrong RC (2008) Managing Scientific Software Complexity with Bocca and CCA. Sci Program 16(4):315–327, DOI 10.3233/SPR-2008-0270

[3] Almgren A, Bell J, Colella P, Howell L, Welcome M (1998) A Conservative Adaptive Projection Method for the Variable Density Incompressible Navier-Stokes Equations. J Comput Phys 142:1–46

[4] AmrLib Homepage (Accessed October 2009) URL `https://ccse.lbl.gov/Software/index.html`

[5] AMROC Homepage (Accessed October 2009) URL `http://amroc.sourceforge.net/`

[6] Anthonissen MJH, Bennett BAV, Smooke MD (2005) An Adaptive Multilevel Local Defect Correction Technique with Application to Combustion. Combust Theory Modelling 9(2):273–299

[7] Arya S, Mount DM, Netanyahu NS, Silverman R, Wu AY (1998) An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. J Assoc Comput Mach 45(6):891–923

[8] Banks JW, Schwendeman DW, Kapila AK, Henshaw WD (2007) A High-Resolution Godunov Method for Compressible Multi-Material Flow on Overlapping Grids. J Comput Phys 223:262–297

[9] Barad M, Colella P (2005) A Fourth-Order Accurate Local Refinement Method for Poisson's Equation. J Comput Phys 209:1–18

[10] Bell JB, Day MS, Shepherd IG, Johnson M, Cheng RK, Grcar JF, Beckner VE, Lijewski MJ (2005) Numerical Simulation of a Laboratory-Scale Turbulent V-flame. Proc Natl Acad Sci USA 102(29):10,006–10,011

[11] Bell JB, Day MS, Grcar JF, Lijewski MJ, Driscoll JF, Filatyev SF (2007) Numerical Simulation of a Laboratory-Scale Turbulent Slot Flame. Proc Combust Inst 31:1299–1307

[12] Bennett BAV, Smooke MD (1998) Local Rectangular Refinement with Application to Axisymmetric Laminar Flames. Combust Theory Modelling 2:221–258

[13] Berger M, Colella P (1989) Local Adaptive Mesh Refinement for Shock Hydrodynamics. J Comput Phys 82:64–84

[14] Bernholdt DE, Allan BA, Armstrong R, Bertrand F, Chiu K, Dahlgren TL, Damevski K, Elwasif WR, Epperly TGW, Govindaraju M, Katz DS, Kohl JA, Krishnan M, Kumfert G, Larson JW, Lefantzi S, Lewis MJ, Malony AD, McInnes LC, Nieplocha J, Norris B, Parker SG, Ray J, Shende S, Windus TL, Zhou S (2006) A Component Architecture for High-Performance Scientific Computing. Intl J High-Perf Computing Appl 20:162–202

[15] Cactus Homepage (Accessed October 2009) URL http://en.wikipedia.org/wiki/Cactus_Framework
[16] CCA Tutorials Hands-On Guide (Accessed October 2009) URL http://www.cca-forum.org/tutorials/
[17] CHOMBO – Infrastructure for Adaptive Mesh Refinement (Accessed October 2009) http://seesar.lbl.gov/anag/chombo/
[18] Cirak F, Deiterding R, Mauch SP (2006) Large-Scale Fluid-Structure Interaction Simulation of Viscoplastic and Fracturing Thin Shells Subjected to Shocks and Detonations. Comput Struct 85(11-14):1049–1065
[19] Cohen SD, Hindmarsh AC (1996) CVODE, a Stiff/Nonstiff ODE Solver in C. Comput Phys 10(2):138–143
[20] Colella P, Graves DT, Keen BJ, Modiano D (2006) A Cartesian Grid Embedded Boundary Method for Hyperbolic Conservation Laws. J Comput Phys 211:347–366
[21] Colella P, Dorr M, Hittinger J, Martin DF, McCorquodale P (2009) High-Order Finite-Volume Adaptive Methods on Locally Rectangular Grids. J Phy: Conf Ser 180:012,010 (5pp), URL http://stacks.iop.org/1742-6596/180/012010
[22] CORBA Component Model Webpage (Accessed October 2009) http://www.omg.org
[23] Dahlgren T, Epperly T, Kumfert G, Leek J (2005) Babel User's Guide. CASC, Lawrence Livermore National Laboratory, Livermore, CA, babel-0.11.0 edn, URL http://www.llnl.gov/CASC/components/docs/users_guide.pdf
[24] Day MS, Bell JB (2000) Numerical Simulation of Laminar Reacting Flows with Complex Chemistry. Combust Theory Modelling 4:535–556
[25] Deiterding R (2000) Object-Oriented Design of an AMR Algorithm for Distributed Memory Computers. In: 8th Int. Conf. on Hyperbolic Problems, Magdeburg
[26] Deiterding R (2005) Detonation Structure Simulation with AMROC. In: Yang LT (ed) High Performance Computing and Communications, no. 3726 in Lecture Notes in Computer Science, Springer, Berlin Heidelberg, pp 916–927
[27] Deiterding R (2009) A Parallel Adaptive Method for Simulating Shock-Induced Combustion with Detailed Chemical Kinetics in Complex Domains. Comput Struct 87:769–783
[28] Deiterding R, Radovitzky R, Mauch SP, Noels L, Cummings JC, Meiron DI (2006) A Virtual Test Facility for the Efficient Simulation of Solid Material Response Under Strong Shock and Detonation Wave Loading. Eng Comput 22(3–4):325–347
[29] Deiterding R, Cirak F, Mauch S, Meiron D (2007) A Virtual Test Facility for Simulating Detonation- and Shock-Induced Deformation and Fracture of Thin Flexible Shells. Int J Multiscale Computational Engineering 5(1):47–63
[30] Drake JB, Jones PW, Carr J George R (2005) Overview of the Software Design of the Community Climate System Model. Intl J High-Perf Computing Appl 19(3):177–186, DOI 10.1177/1094342005056094, URL http://

hpc.sagepub.com/cgi/content/abstract/19/3/177, http://hpc.sagepub.com/cgi/reprint/19/3/177.pdf

[31] Dubey A, Antypas K, Ganapathy MK, Reid LB, Riley K, Sheeler D, Siegel A, Weide K (2009) Extensible Component Based Architecture for FLASH, A Massively Parallel, Multiphysics Simulation Code. Parallel Comput Submitted, preprint at http://arxiv.org/pdf/0903.4875

[32] Earth Systems Modeling Framework Homepage (Accessed October 2009) URL http://www.esmf.ucar.edu/

[33] Englander R, Loukides M (1997) Developing Java Beans (Java Series). O'Reilly and Associates, http://www.java.sun.com/products/javabeans

[34] Falgout R, Yang U (2002) Hypre: a Library of High Performance Preconditioners, in Computational Science. In: Sloot PMA, Tan C, Dongarra JJ, Hoekstra AG (eds) Lecture Notes in Computer Science, vol 2331, Springer-Verlag, pp 632–641

[35] Fryxell B, Olson K, Ricker P, Timmes FX, Zingale M, Lamb DQ, MacNeice P, Rosner R, Truran JW, Tufo H (2000) FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes. Ap J Supplement Series 131:273–334

[36] Godfrey MW, Tu Q (2000) Evolution in Open Source Software: A Case Study. In: Proceedings of the International Conference on Software Maintenance, pp 131–142, URL http://citeseer.nj.nec.com/300079.html

[37] Goodale T, Allen G, Lanfermann G, Mass J, Radke T, Seide E, Shalf J (2002) The Cactus Framework and Toolkit: Design and Applications. In: Proceedings of Vector and Parallel Processing - VECPAR 2002

[38] GrACE Homepage (Accessed October 2009) http://www.caip.rutgers.edu/TASSL/

[39] Henshaw WD (1994) A Fourth-Order Accurate Methods for the Incompressible Navier-Stokes Equations on Overlapping Grids. J Comput Phys 113:13–25

[40] Henshaw WD, Schwendeman DW (2003) An Adaptive Numerical Scheme for High-Speed Reactive Flow on Overlapping Grids. J Comput Phys 191:420–447

[41] Henshaw WD, Schwendeman DW (2006) Moving Overlapping Grids with Adaptive Mesh Refinement for High-Speed Reactive and Non-reactive Flow. J Comput Phys 216:744–779

[42] Henshaw WD, Schwendeman DW (2008) Parallel Computation of Three-Dimensional Flows using Overlapping Grids with Adaptive Mesh Refinement. J Comput Phys 227:7469–7502

[43] Huelette GC, Sottile MJ, Armstrong R, Allan B (2009) OnRamp: Enabling a New Component-Based Development Paradigm. In: Proceedings of Component-Based High Performance Computing

[44] Kadioglu S, Klein R, Minion M (2008) A Fourth-Order Auxiliary Variable Projection Method for Zero-Mach Number Gas Dynamics. J Comput Phys 227:2012–2043

[45] Kenny JP, Janssen CL, Valeev EF, Windus TL (2008) Components for Integral Evaluation in Quantum Chemistry. J Comput Chem 29(4):562–577
[46] Krishnan M, Alexeev Y, Windus TL, Nieplocha J (2005) Multilevel Parallelism in Computational Chemistry using Common Component Architecture and Global Arrays. In: SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, IEEE Computer Society, Washington, DC, USA, p 23, DOI http://dx.doi.org/10.1109/SC.2005.46
[47] Lam S (1993) Using CSP to Understand Complex Chemical Kinetics. Combust Sci Technol 89:375–404
[48] Lam S, Goussis D (1988) Understanding complex chemical kinetics with computational singular perturbation. Proc Combust Inst 22:931–941
[49] Lee J, Najm H, Lefantzi S, Ray J, Goussis D (2005) On Chain Branching and its Role in Homogeneous Ignition and Premixed Flame Propagation. In: Bathe K (ed) Computational Fluid and Solid Mechanics 2005, Elsevier Science, pp 717–720
[50] Lee J, Najm H, Lefantzi S, Ray J, Frenklach M, Valorani M, Goussis D (2007) A CSP and Tabulation Based Adaptive Chemistry Model. Combustion Theory and Modeling 11(1):73–102
[51] Lee JC, Najm HN, Lefantzi S, Ray J, Frenklach M, Valorani M, Goussis D (2007) A CSP and Tabulation Based Adaptive Chemistry Model. Combust Theory Modelling 11(1):73–102
[52] Lefantzi S, Ray J, Najm HN (2003) Using the Common Component Architecture to Design High Performance Scientific Simulation Codes. In: Proceedings of the International Parallel and Distributed Processing Symposium, Nice, France
[53] Lefantzi S, Ray J, Kennedy CA, Najm HN (2005) A Component-based Toolkit for Reacting Flows with High Order Spatial Discretizations on Structured Adaptively Refined Meshes. Prog Comput Fluid Dy 5(6):298–315
[54] Li X, Parashar M (2004) Hierarchical Partitioning Techniques for Structured Adaptive Mesh Refinement Applications. J Supercomput 28(3):265–278
[55] Li X, Parashar M (2007) Hybrid Runtime Management of Space-Time Heterogeneity for Parallel Structured Adaptive Applications. IEEE Transactions on Parallel and Distributed Systems 18(9):1202–1214
[56] Liu H, Parashar M (2005) Enabling Self-management of Component-based High-Performance Scientific Applications. In: Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14), Research Triangle Park, NC
[57] Liu H, Parashar M (2006) Accord: A Programming Framework for Autonomic Applications. IEEE Transaction on Systems, Man, and Cybernetics 36(3):341–352, special issue on Engineering Autonomic Systems, Editors: R. Sterritt and T. Bapty
[58] Majda A, Sethian J (1985) The Derivation and Numerical Solution of the Equations for Zero Mach Number Combustion. Comb Sci Technology 42:185–205

[59] Martin DF, Colella P (2000) A Cell-Centered Adaptive Projection Method for the Incompressible Euler Equations. J Comput Phys 163:271–312
[60] McBride BJ, Gordon S, Reno M (1993) Coefficients for Calculating Thermodynamic and Transport Properties of Individual Species. Tech. Rep. TM-4513, NASA
[61] McInnes LC, Allan BA, Armstrong R, Benson SJ, Bernholdt DE, Dahlgren TL, Diachin LF, Krishnan M, Kohl JA, Larson JW, Lefantzi S, Nieplocha J, Norris B, Parker SG, Ray J, Zhou S (2006) Parallel PDE-Based Simulations Using the Common Component Architecture. In: Numerical Solution of Partial Differential Equations on Parallel Computers, Springer, pp 327–384, also available as ANL/MCS-P1179-0704 via http://www.mcs.anl.gov/cca/publications/p1179.pdf
[62] McInnes LC, Ray J, Armstrong R, Dahlgren TL, Malony A, Norris B, Shende S, Kenny JP, Steensland J (2006) Computational Quality of Service for Scientific CCA Applications: Composition, Substitution, and Reconfiguration. Tech. Rep. ANL/MCS-P1326-0206, Argonne National Laboratory, URL ftp://info.mcs.anl.gov/pub/tech_reports/reports/P1326.pdf
[63] Meir "Manny" Lehman's FEAST project (Accessed October 2009) URL http://www.doc.ic.ac.uk/~mml/feast
[64] Najm H, Knio O (2005) Modeling Low Mach Number Reacting Flow with Detailed Chemistry and Transport. J Sci Comp 25(1):263–287
[65] Overture Homepage (Accessed October 2009) URL https://computation.llnl.gov/casc/Overture/
[66] Pantano C, Deiterding R, Hill DJ, Pullin DI (2007) A Low Numerical Dissipation Patch-Based Adaptive Mesh Refinement Method for Large-Eddy Simulation of Compressible Flows. J Comput Phys 221(1):63–87
[67] Parashar M, Browne JC (2000) System Engineering for High Performance Computing Software: The HDDA/DAGH Infrastructure for Implementation of Parallel Structured Adaptive Mesh Refinement. In: S B Baden DBG M P Chrisochoides, Norman ML (eds) Structured Adaptive Mesh Refinement, IMA, vol 117, Springer-Verlag
[68] Paul PH (1997) DRFM: A New Package for the Evaluation of Gas-Phase-Transport Properties. Sandia Report SAND98-8203, Sandia National Laboratories, Albuquerque, New Mexico
[69] Publications from the Applied Numerical Algorithms Group (Accessed October 2009) URL http://seesar.lbl.gov/anag/publication.html
[70] Publications Using AMROC and Virtual Test Facility (Accessed October 2009) URL http://www.csm.ornl.gov/~r2v/html/pub.htm
[71] Ray J, Najm HN, Milne RB, Devine KD, Kempka S (2000) Triple Flame Structure and Dynamics at the Stabilization Point of an Unsteady Lifted Jet Diffusion Flame. Proc Combust Inst 28:219–226

[72] Ray J, Kennedy C, Steensland J, Najm HN (2005) Advanced Algorithms for Computations on Block-Structured Adaptively Refined Meshes. J Phys: Conf Ser 16:113–118
[73] Ray J, Kennedy CA, Lefantzi S, Najm HN (2007) Using High-Order Methods on Adaptively Refined Block-Structured Meshes - Derivatives, Interpolations, and Filters. SIAM J Sci Comp 29(1):139–181
[74] Safta C (2009) Personal Communication
[75] Safta C, Ray J, Najm H (2009) A High-Order Projection Scheme for AMR Computations of Chemically Reacting Flows. In: Proceedings of the 2009 Fall Meeting of the Western States Section of the Combustion Institute, Irvine, CA, URL http://www.caip.rutgers.edu/~jaray/
[76] Sommeijer BP, Shampine LF, Verwer JG (1998) RKC: An Explicit Solver for Parabolic PDEs. J Comp Appl Math 88:315–326
[77] de St Germain JD, McCorquodale J, Parker SG, Johnson CR (2000) UINTAH: A Massively Parallel Problem Solving Environment. In: HPDC '00 : Ninth IEEE International Symposium on High Performance and Distributed Computing
[78] van Straalen B, Shalf J, Ligocki T, Keen N, Yang WS (2009) Scalability Challenges for Massively Parallel AMR Applications. In: Proceedings of the 23rd IEEE International Symposium on Parallel and Distributed Processing, URL https://seesar.lbl.gov/ANAG/publication.html
[79] TAU: Tuning and Analysis Utilities (Accessed November 2009) Http://www.cs.uoregon.edu/research/paracomp/tau/
[80] The OpenFOAM Homepage (Accessed October 2009) URL http://www.opencfd.co.uk/openfoam/
[81] Tonse S, Moriarty N, Brown N, Frenklach M (1999) PRISM: Piecewise Reusable Implementation of Solution Mapping. An economical strategy for chemical kinetics. Israel Journal of Chemistry 39:97–106
[82] Trebon N, Morris A, Ray J, Shende S, Malony AD (2007) Performance Modeling Using Component Assemblies. Concurr Comp-Pract E 19(5):685–696
[83] Valorani M, Goussis D (2001) Explicit Time-Scale Splitting Algorithm For Stiff Problems: Auto-Ignition Of Gaseous-Mixtures Behind A Steady Shock. J Comput Phys 169:44–79
[84] Verwer JG, Sommeijer BP, Hundsdorfer W (2004) RKC Time-stepping for Advection-Diffusion-Reaction Problems. J Comput Phys 201(1):61–79, DOI http://dx.doi.org/10.1016/j.jcp.2004.05.002
[85] Visual Basic Webpage (Accessed October 2009) http://msdn.microsoft.com/en-us/vbasic/default.aspx
[86] Williams F (1985) Combustion Theory, 2nd edn. Addison-Wesley, New York
[87] XCAT Homepage (Accessed October 2009) http://www.extreme.indiana.edu/xcat/
[88] Yetter R, Dryer F, Rabitz H (1991) A Comprehensive Reaction Mechanism for Carbon Monoxide/Hydrogen/Oxygen Kinetics. Combust Sci Technol 79:97