Using the Common Component Architecture to Design High Performance Scientific Simulation Codes

Sophia Lefantzi, Jaideep Ray and Habib N. Najm
Sandia National Laboratories
PO Box 969, MS 9051, Livermore, CA, 94550
{slefant,jairay,hnnajm}@ca.sandia.gov

Abstract

We present a design and proof-of-concept implementation of a component-based scientific simulation toolkit for hydrodynamics. We employed the Common Component Architecture, a minimalist, low-latency component model as our paradigm for developing a set of high-performance parallel components for simulating flows on structured adaptively refined meshes. Our findings demonstrate that the architecture is sufficiently flexible and simple to allow an intuitive and straightforward decomposition of a complex monolithic code into easy-to-implement components. The result is a set of stand-alone independent components from which a simulation code is assembled. Our results show that the component architecture imposes negligible overheads on single processor performance while scaling to multiple processors remains unaffected.

Keywords: Common Component Architecture (CCA), Structured Adaptive Mesh Refinement (SAMR), component-based software, high performance computing, hydrodynamics.

1. Introduction

Most scientific codes today are hand-tooled by a small group, and the usual practice is to embody all numerical and domain-specific algorithms in a monolithic simulation code. The small group is responsible for maintaining and developing the code, a fact that restricts the size and the scope of parallel simulations to the abilities and expertise of a few people. A detailed discussion of the software issues in scientific simulations can be found in [7]. An obvious answer to the problem of maintaining, reusing and exchanging scientific code developed by experts is the creation of specialized, individual scientific modules.

The scientific community addressed the modularity issue by developing individual libraries of special purpose, widely used algorithms. Libraries like the Basic Linear Algebra Library (BLAS), the Linear Algebra Package (LAPACK), and the Message Passing Interface (MPI) provide very specific computing solutions and are available on all high performance platforms. Libraries have a well defined, standardized interface but their actual implementations are vendor-specific. The user follows the standard; once linked, the libraries "just work". Another approach to modularity was achieved through object-oriented programming with the development of frameworks (*e.g.* POOMA [18], OVERTURE [12]) for parallel scientific computing. These frameworks provide a set of data-structures and a large set of commonly used numerical algorithms. Developing codes using the framework's own algorithms and data-structures is fast and usually straightforward; however one's codes are tightly coupled to the framework and are no longer independent. There are less intrusive frameworks, *e.g.* GrACE [23] and Uintah[15], which only serve as data managers. They have a data model and an interface for other codes to access the data. Orchestration and synchronization of the work is left to the user.

A valuable pointer for achieving modularity in scientific computing comes from commercial practices. The business world has implemented modularity by adopting the *component* model (*e.g.* Visual Basic [5], CORBA [1] and Java Beans [16]). In this model an object implements a functionality, which is exploited via interfaces; an object's adherence to a specification (dictated by the component model) transforms it into a component within the framework. Components are peers, *i.e.* they do not inherit from other components, and are easily extensible since components implementing an agreed-to, well defined interface can be developed in complete isolation. While component-based software design enhances the co-operative development of applications, the commercial model is unsuitable for scientific computing [7], the main drawbacks being high latency and lack of support for parallel (not distributed) computing. The CCA (Common Component Architecture) component model was designed [8] to meet the high performance requirements of scientific computing; to date three CCA-compliant framework implementations have been demonstrated (CCAFFEINE[7], Uintah[15] and XCAT[6]). The CCA standard is flexible; while allowing an evolutionary path forward for new scientific applications, it is also an efficient way of "wrapping" legacy codes. An interesting discussion of the latter along with simple illustrative problems (*e.g.* Laplace equation) can be found in [22].

The present work is the first comprehensive account of how one identifies a class of significant computational scientific problems (involving realistic physical models, nonlinear PDEs and a spectrum of time and length scales) and creates a CCA-compliant component-based software infrastructure to solve them. This infrastructure consists of *scientific components*, where each component has a distinct functionality in terms of physical modeling or implementation of a numerical algorithm. In this paper we study and lay out a formalism (the "classical" approach [14]) by which one identifies a system and decomposes it recursively to a component-level granularity, thus identifying the solution architecture. We then create these components and assemble them to study two very different problems; a reacting flow problem with two different test cases (0D ignition and 2D reaction-diffusion) and a shock hydrodynamics problem. Despite the different physical nature, both problems adhere to the mathematical abstraction (system) that the software infrastructure is expected to solve. Our domain of interest is hydrodynamics, and the mathematical abstraction of the physical problems we target is a set of nonlinear partial differential equations. We discuss the physical model for each of the two problems, the associated component assemblies, and present computational results highlighting the performance of the component-code in terms of overhead and parallel scalability.

2. The CCA Component Model

The CCA model [8] uses the *provides-uses* design pattern. Components *provide* functionalities through interfaces that they export; they *use* other components' functionalities via interfaces. These interfaces are called *Ports*; thus a component has ProvidesPorts and UsesPorts. Components are peers and are independent. They are created and exist inside a framework; this is where they register themselves, declare their UsesPorts and ProvidesPorts and connect with other components.

CCAFFEINE [7] is the CCA framework we employ for our research. CCAFFEINE is a low latency framework for scientific computations. Components can be written in most languages within the framework; we develop most of our components in C++ or as sets of F77 libraries wrapped in C++. All CCAFFEINE components are derived from a data-less abstract class with one deferred method called *setServices(Services *q)*. All components implement the *setServices* method which is invoked by the framework at component creation and is used by the components to register themselves and their UsesPorts and ProvidesPorts. Components also implement other data-less abstract classes, called Ports, to allow access to their standard functionalities. Every component is compiled into a shared object library, *i.e.* a dynamically loadable library. Most Ports are domain-specific and their design is left to the user community.

A CCAFFEINE code can be assembled and run through a script or a Graphical User Interface (GUI). All components exist on the same processor and the same address space. Once components are instantiated and registered with the framework, the process of connecting ports is just the movement of (pointers to) interfaces from the *providing* to the *using* component. A method invocation on a UsesPort thus incurs a virtual function call overhead before the actual implemented method is used. CCAFFEINE uses the SCMD (Single Component Multiple Data) [7] model of parallel computation. Identical frameworks, containing the same components, are instantiated on all P processors. "Parallelness" is implemented by the components via MPI communications between the same component on all P processors, *i.e.* across all P processors, P instances of a given component X form a *cohort*[7] within which all message passing is done. Thus the framework does not provide any message-passing services. The framework adheres to the MPI-1 standard, *i.e.* dynamic process creation/deletion and a dynamically sized parallel virtual machine are not supported. This minimalist nature renders CCAFFEINE light, simple, fast, and very unobtrusive to the components. Performance is left to the component developer who is in the best position to determine the optimal algorithms and implementations for the problem at hand.

A CCAFFEINE job is generally started using *mpirun* (or equivalent). "P" instances of the framework, run with the same script, cause P identically configured frameworks to load and exist on as many processors. The framework lends out a properly scoped MPI communicator to any component to allow access to the parallel virtual machine created by mpirun. This conventional mode of starting a component-based code makes job submission to a queuing system rather easy. CCAFFEINE can also be started via a GUI. The framework comes with an application framer which allows the user to compose an application by dragging and dropping components from an available list into an "arena" and connecting the appropriate ports. Fig. 1 shows a small code assembled in the arena – components are black boxes, with ProvidesPorts on the left, UsesPorts on the right, and lines connecting the Ports. Any action performed in the GUI is converted to the corresponding script command action and fed into a "multiplexer", which reproduces the action P-fold and issues it to each instance of the framework. The output from the framework instances is de-multiplexed and displayed by the GUI. The GUI-to-framework communication employing the multiplexer uses sockets.

3. Problem Specification

Our objective is the creation of a component-based software infrastructure for computational hydrodynamics, including chemically reacting flows. We target scientific problems defined on simple geometries, *i.e.* logically rectangular domains. Typically a mesh of rectangular cells is overlaid on the domain and flow quantities (density, momentum in x, y, z directions, temperature, species concentrations, *etc.*) are defined at the cell corners (henceforth called mesh or grid points). Evolution in time of these quantities is governed by PDEs which are of the general form:

$$\frac{\partial \mathbf{\Phi}}{\partial t} = \mathbf{F}(\mathbf{\Phi}, \nabla\mathbf{\Phi}, \nabla^2\mathbf{\Phi}, ...) + \mathbf{G}(\mathbf{\Phi}) \tag{1}$$

where $\mathbf{\Phi}$ is the vector of flow variables at a given mesh point. Note that $\mathbf{G}$ involves variables only at a mesh point while $\mathbf{F}$ involves spatial derivatives which are computed using finite difference or finite volume schemes [17] and consequently depend upon the mesh point and its close neighbors. For different physical problems, $\mathbf{\Phi}$, $\mathbf{F}$ and $\mathbf{G}$ vary. In some cases $\mathbf{F}$ or $\mathbf{G}$ may be absent.

Often $\mathbf{\Phi}$ exhibits steep spatial variations in scattered time-evolving regions of the domain. The dependence of $\mathbf{F}$ on neighboring points requires that these steep variations be fully resolved, usually by locally increasing the grid density. As the flowfield evolves, the regions of high gradients move and the locally-resolved grid adapts. One of the most common techniques of adaptive grid refinement is Structured Adaptive Mesh Refinement (SAMR) [10]. As a first step, a uniform coarse mesh is overlaid on the domain. The coarseness of the mesh causes errors (suitably defined) in regions of high gradients. Based on an error threshold, grid points in these regions are flagged, collated into rectangles, and finer meshes are created by dividing the coarse cells symmetrically by a constant refinement factor. This occurs recursively, leading to a hierarchy of patches. The details are in [23]. The patch hierarchy is periodically recreated. The solution is passed through a filter to determine regions needing finer meshes, whereby new patches are created and initialized with data from the coarse meshes (provided there does not exist a patch of the same resolution over that subdomain, wholly or partly). This process is called *prolongation*. Regions which are deemed over-refined have fine patches destroyed. Upon patch recreation the domain decomposition on multiple processors is re-defined. An initial $\mathbf{\Phi}$ is imposed on all patches (Initial Condition) and the system is evolved in time by integrating over time-steps. This also resolves temporal changes in $\mathbf{\Phi}$. Each time-step is assumed to advance in time from t^n to $t^n + \Delta t = t^{n+1}$.

We assume that $\mathbf{G}$, in Eq. 1, is stiff (the ratio of the largest and the smallest eigenvalues of $\partial\mathbf{G}/\partial\mathbf{\Phi}$ is large) while $\mathbf{F}$ is non-stiff. We employ an operator-splitting [28] technique. Below, we outline the design of a CCA-component based infrastructure that enables the solution of equations like Eq. 1.

4. Design of Components and Interfaces

In this section we demonstrate the rationale for the recursive decomposition of the solution strategy of Eq. 1 into software subsystems and the interfaces developed. A *software subsystem* is a collection of components that embodies a physical or numerical functionality (*e.g.,* an **Explicit integration subsystem** includes the time integrator, the RHS evaluator and miscellaneous components that identify the largest eigenvalue of the discretization matrix to enable dynamic time-step sizing). The software subsystems we identified are:

1. **Mesh:** It serves as a means of declaring and maintaining patches in the mesh hierarchy. It is geometric in nature, and determines and administers the child-parent-sibling relationships and the spatio-temporal location of patches. Load balancing and domain decomposition functionalities are implemented here.

2. **Data Object:** It maintains the collection of arrays which contain data declared on patches, 1 array per patch. Typically a number of related variables are stored together in a Data Object; equally typically, a simulation would contain 2-3 Data Objects. This subsystem implements the actual movement/copying of data between patches and the packing/unpacking of data before/after message passing. Currently we have wrapped GrACE [3, 23] into a C++ component to perform the **Data Object** and the **Mesh** tasks.

3. **Initial Condition:** This subsystem consists of a set of components that impose Initial Conditions on a **Data Object**.

4. **Explicit Integration subsystem:** It consists of a recursive time integrator that advances a set of Data Objects over a time step as well as components that evaluate and assemble the Right Hand Side (RHS), one patch at a time. The evaluation of the RHS can be done by one component or by a further subsystem of components. This also contains components that analyze the field to determine an approximation of the highest eigenvalue that the integrator will encounter. This information is used by the integrator to dynamically adjust the timestep.

5. **Implicit Integration subsystem:** This consists of an implicit time integrator, which advances a vector of variables, RHS component(s), and an adaptor that collates data from a patch to a vector.

6. **Interpolation components:** These implement various spatial and temporal interpolation operators.

7. **Boundary Condition:** It is applied on a patch by patch basis. BCs are applied at each of the stages of a multi-stage integration scheme; hence application of the boundary conditions has to be done on a finer basis than one Data Object at a time. Thus the granularity will be a patch.

8. **Database components:** These components store certain parameters (*e.g.* mesh size, gas properties, *etc*), that are retrieved using a key-value pair mechanism. They are essentially maps between the (character string) property name and a number.

9. **Adaptors:** Depending on the physical problem at hand, case-specific adaptors are often used to consolidate and filter outputs from various physics components.

Given the functional description above, it is clear what types of Ports (interfaces) are needed : (a) Port(s) (provided by the mesh component) that allow (i) geometrical manipulation of the domain, (ii) the declaration of fields on the mesh (via **Data Objects**), and (iii) allow tasks like setting/querying of domain-decomposition details. Our design for type (a) Ports is called **MeshPort** [4]. (b) An abstract interface for the **Data Object** allowing manipulation of patches and the data defined on them. (c) Ports that accept an array of **Data Objects** and act on them in a synchronized manner. Integrators usually support these ports. (d) Ports that accept an array from a patch. (e) Ports that accept vectors. (f) Ports that allow setting/querying of key-value pairs.

In this section we will show the use and reuse of a set of components developed for solving mathematical systems like Eq. 1. These relate, physically, to a 0D/homogeneous ignition, ignition in a two-dimensional (2D) reaction-diffusion system and a shock interacting with a density inhomogeneity modeled using the Euler equations (an approximation of shock-induced mixing of two gases).

4.1. Zero Dimensional Ignition Problem

The 0D ignition problem is described by the system:

$$\frac{d\mathbf{\Phi}}{dt} = \mathbf{G}(\mathbf{\Phi}) \qquad (2)$$

where $\mathbf{\Phi} = \{T, Y_1, ..., Y_{N-1}, P_0\}$, T is the gas temperature, Y_i is the mass fraction of species i in the mixture, P_0 is the stagnation pressure, N is the total number of species, and $\mathbf{G}$ is the array of chemical source terms for $\mathbf{\Phi}$ computed from the heat equation, the species equations and the conservation of mass. Eq. 2 is identical to Eq. 1, without the spatial derivative term $\mathbf{F}$. We use a H_2–Air mechanism with 9 species and 19 reversible reactions [26]. This set of equations is a 0D reduction of the low Mach number form of the Navier-Stokes, energy, and species equations; details about the equations can be found in [20, 21].

The solution of this system requires the following modules (a) Initial Condition (b) Stiff Integrator (c) RHS Evaluator and (d) Boundary Condition. These modules are implemented as the (a) **Initialize**, (b) **CvodeComponent**, (c) **ThermoChemistry**, (d) **ProblemModeler** and **dPdt** components respectively. Fig. 1 shows the 0D ignition code as assembled in the CCAFFEINE framework GUI. The component design mapping to the description in Sec. 4 is in Table 1.

Software Subsystems	Component Instance
Mesh	N/A
Data Object	N/A
Initial Condition	Initializer
Explicit Integration	N/A
Implicit Integration	CvodeComponent, ThermoChemistry
Boundary Condition	problemModeler, dPdt
Database	ThermoChemistry
Adaptors	problemModeler

Table 1. Component Design for the 0D ignition code.

Component **Initializer** imposes the initial condition; a vector of double precision numbers specifying the stoichiometric mass fractions for the species, the initial temperature (1000 K), and the initial pressure (1 atm). The Implicit Integration subsystem consists of **CvodeComponent** and **ThermoChemistry** components. **CvodeComponent** is an implicit stiff/non-stiff integrator that time-advances the system as it ignites. This is a thin wrapper around the Cvode [13] integrator library. The **ThermoChemistry** component embodies the chemical interactions; it provides the source terms for temperature and species due to chemistry and is a thin C++ wrapper around Fortran 77 subroutines abstracted from pre-existing codes for chemically reacting flows. **ThermoChemistry** also serves as a Database subsystem, *i.e.* it holds the gas properties. Between **CvodeComponent** and **ThermoChemistry** is the **problemModeler** component which acts as an Adaptor, *i.e.* for this closed system it adds the pressure term to the heat equation. The pressure term depends on the boundary conditions of the problem (rigid walls, *i.e.* constant mass and volume) and is computed by the **dPdt** component. The code integrates up to 1 ms and executes in 1.5 secs on a 1 GHz Pentium III Red Hat workstation.

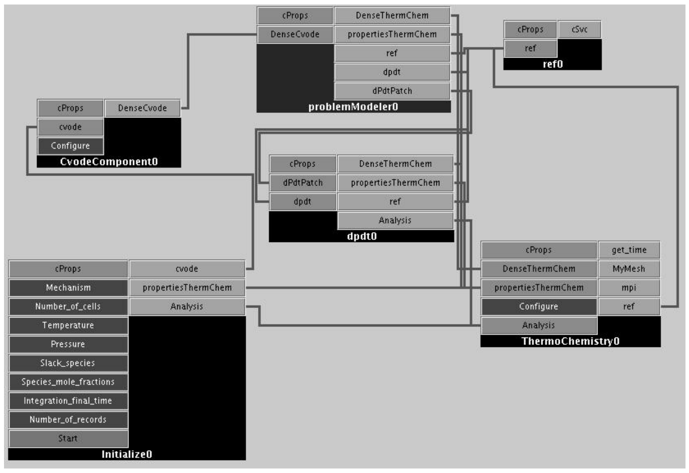

Figure 1. GUI shot of the 0D ignition code.

4.2. 2D Reaction-Diffusion Fronts

In this example we expand the 0D ignition test case to include spatial terms to model diffusion. The equations are:

$$\frac{\partial \mathbf{\Phi}}{\partial t} = \mathbf{K}\nabla \cdot (\mathbf{B}\nabla\mathbf{\Phi}) + \mathbf{R} \tag{3}$$

where $\mathbf{\Phi} = \{T, Y_1, ..., Y_{N-1}\}$, $\mathbf{K} = \frac{1}{\rho}\{\frac{1}{c_p}, 1, ..., 1\}$, $\mathbf{B} = \{\lambda, \rho D_1, ..., \rho D_{N-1}\}$, $\mathbf{R}$ is the reactive production of heat and species, while $\mathbf{K}\nabla \cdot (\mathbf{B}\nabla\mathbf{\Phi})$ is the diffusive transport source term (by Fick's Law) of the heat and the species. λ is the thermal conductivity and D_i are the diffusion coefficients. The chemical production of a species is governed by the chemical reactions it undergoes; heat production (by chemical reactions) is governed by the summation over all the species of the enthalpy change of each species. This system of partial differential equations models a Reaction-Diffusion system.

This is a reaction-diffusion flame model that includes chemistry and the diffusion of heat and species, but no convection; pressure is assumed to be constant in time and space (*i.e.* burning in an open domain). The species are assumed to diffuse independently into the mixture at a mesh point, *i.e.* the diffusion coefficient D_i of the i^{th} species is mixture averaged. The model was developed to study the behavior of an actual flame simulation under SAMR in a component based infrastructure. We used the same H_2–Air mechanism we used for the 0D ignition test case. The diffusion of species provides a good approximation of the "predictable" part of the workload of a real flame simulation, *i.e.* the compute time per mesh point at refinement level l can be predicted and is uniformly applicable across all mesh points. Chemistry integration in our reaction-diffusion example is expensive mostly inside the flame, and contributes tremendously to load-imbalance. Patches are collated and distributed among processors to maximize load-balance while keeping parents and children on the same processors. Fig. 2 shows the component assembly design described in Table 2. The **InitialCondition** component initializes a configuration with three hot-spots. The Mesh, Data Object and Boundary Condition subsystems are accommodated by **GrACEComponent**, which is the componetized version [4] of the GrACE library. The Implicit Integration subsystem is identical to that of the 0D ignition code. The **ImplicitIntegrator** component is an Adaptor that calls on the Implicit Integration subsystem for all cells and all patches. The Explicit Integration subsystem consists of three components: a Runge-Kutta-Chebyshev integrator [9] (**ExplicitIntegrator**), a component to calculate the diffusion fluxes (**DiffusionPhysics**) and a component that calculates the mixture-averaged diffusion coefficients (**DRFMComponent**). **DRFMComponent** is a thin C++ wrapper around the Fortran77 DRFM [24] package. **(MaxDiffCoeffEvaluator)** component is used by the explicit integrator to evaluate the maximum diffusion coefficient over the domain to determine the maximum stable timestep. **ErrorEstAndRegrid**) component estimates the gradients at a cell and flags regions for refinement/coarsening. The code runs for 58 hours on 28 CPUs (Beowulf cluster, Red Hat 7.2, 1 GHz, 512 kB cache, Pentium III processors, 1 GB RAM per node, 2 CPUs per node) connected by 100 bT switched fast Ethernet.

Software Subsystems	**Component Instance**
Mesh	GrACEComponent
Data Object	GrACEComponent
Initial Condition	InitialCondition
Explicit Integration	ExplicitIntegrator, DiffusionPhysics, DRFMComponent
Implicit Integration	CvodeComponent, ThermoChemistry
Boundary Condition	GrACEComponent
Database	ThermoChemistry
Adaptors	ImplicitIntegrator

Table 2. Component Design for the 2D Reaction-Diffusion code.

The simulation is done within a 10 mm square domain.

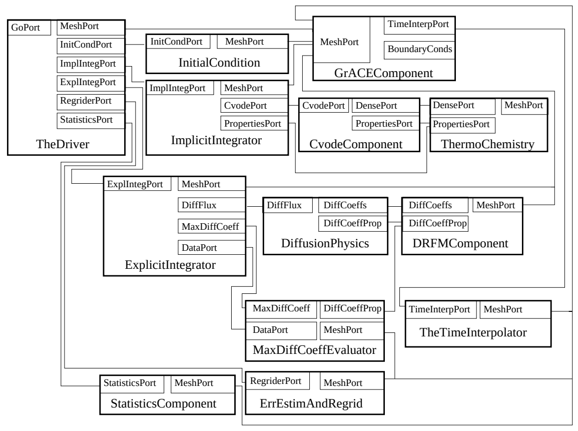

Figure 2. Reaction-Diffusion code assembly.

A 100×100 coarse mesh is laid on the domain. A stoichiometric $H_2 - $ Air mixture is defined on the square with three hot-spots which ignite. In Fig. 3 we plot the evolution of the temperature. The finest structures are around 100×10^{-3} mm in size; the finest grid resolution is 12×10^{-3} mm. The refinement ratio is 2. The spectrum of length scales, spanning two orders of magnitude, is resolved using SAMR as shown in Fig. 4.

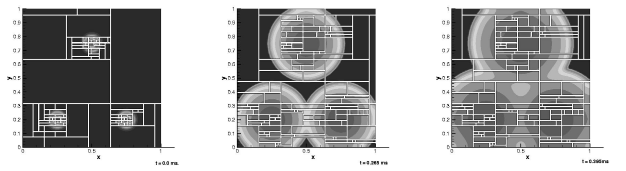

Figure 3. Temperature field at t = 0, 0.265, 0.395 ms.

4.3. 2D Shock-Interface Interaction

In this example, we show reuse of the components mentioned above in the interaction of a shock with a density interface. The system is modeled using the 2D Euler equation (inviscid Navier-Stokes); details of the equations used and the interaction are in [27]. The governing equations (compressible Euler equations) in conservative form are:

$$\mathbf{U}_t + \mathcal{F}(\mathbf{U})_x + \mathcal{G}(\mathbf{U})_y = 0 \tag{4}$$

where

$$\mathbf{U} = \{\rho, \rho u, \rho v, \rho e, \rho\zeta\}^T,$$

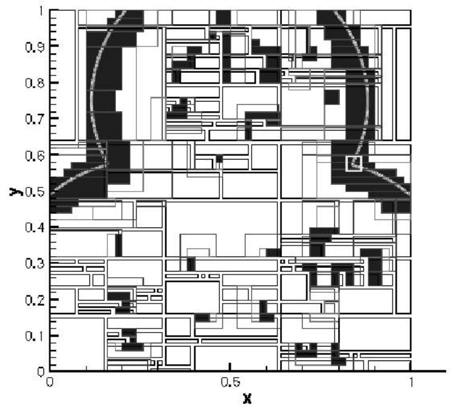

Figure 4. AMR patch distribution with $\mathbf{H_2O_2}$ mass fraction plotted on the finest mesh.

$$\begin{aligned}\mathcal{F}(\mathbf{U}) &= \{\rho u, \rho u^2 + p, \rho uv, (\rho e + p)u, \rho\zeta u\}^T,\\ \mathcal{G}(\mathbf{U}) &= \{\rho v, \rho uv, \rho v^2 + p, (\rho e + p)v, \rho\zeta v\}^T,\end{aligned}$$

ρe is the total energy, related to the pressure p by $p = (\gamma - 1)(\rho e - \frac{1}{2}\rho(u^2 + v^2))$ and ζ is a interface tracking function. We use the ideal gas law as the equation of state.

The equations are solved on a 3 level adaptive mesh, using a finite volume Godunov method [27]. The mesh is cell-centered, *i.e.* the mesh divides the domain into small rectangular cells and fluid variables are defined and indexed at the cell centers. The Godunov method involves constructing the states on the left and right of a cell interface using slope-limiters, upwinding and solving a Riemann problem [27]. The construction of left and right states holds true for most finite volume methods; solving an exact Riemann problem could be substituted by a gas-kinetics scheme (*e.g.* Equilibrium Flux Method [25]).

The code assembly of components (Fig. 5) is described in Table 3. There is a **ConicalInterfaceIC** component that sets up the problem - a shock tube with Air and Freon (density ratio 3) separated by an oblique (30^o from the vertical) interface which is ruptured by a Mach 1.5 shock. The **GrACEComponent**, **StatisticsComponent**, and **ErrorEstAndRegrid** components first utilized in the Reaction-Diffusion simulation (Sec. 4.2, Fig. 2), are being reused. A Runge-Kutta time integrator (**ExplicitIntegratorRK2**) with an **InviscidFlux** component supplies the right-hand-side of the equation, patch-by-patch. **InviscidFlux** component uses a **States** component to set up the Riemann problem at each cell interface which is then passed to the **GodunovFlux** component for the Riemann solution. **CharacteristicQuantities** determines the characteristic speeds and

Software Subsystems	Component Instance
Mesh	GrACEComponent
Data Object	GrACEComponent
Initial Condition	ConicalInterfaceIC
Explicit Integration	ExplicitIntegratorRK2, GodunovFlux, States
Implicit Integration	N/A
Boundary Condition	BoundaryConditions
Database	GasProperties
Adaptors	InviscidFlux

Table 3. Component Design for the 2D Shock-Interface Interaction.

ProlongRestrict performs the cell-centered interpolations. The shock tube has reflecting boundary conditions above and below and outflow on the right, which are set with the **BoundaryConditions** component.

In Fig. 6 we plot the density field after the shock-interface interaction; the thick black line indicates the interface $\zeta = \mathbf{0.5}$. Reflected shocks are seen. Note that regions of steep pressure and density gradients (shock waves and gas-gas interface respectively) are resolved with Level 3 meshes. In Fig. 7 we plot the circulation on the interface ($\Gamma = \int_{0.001 \le \zeta \le 0.999} \vec{\omega} \cdot dA$) as we increase the levels of refinement. We note that we achieve convergence of the interfacial circulation deposition since there is no appreciable difference between the 2-level and 3-level runs. Further, the maximum deposition, corresponding to the "knee" in the plot, is closest to the analytical estimate of -0.592 for the 3-level run.

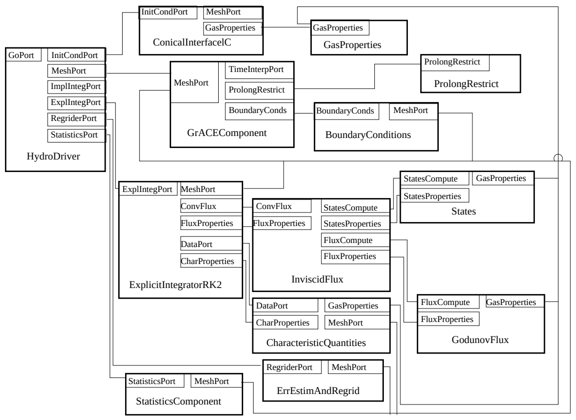

Figure 5. A component assembly to simulate shock interactions with density inhomogeneities.

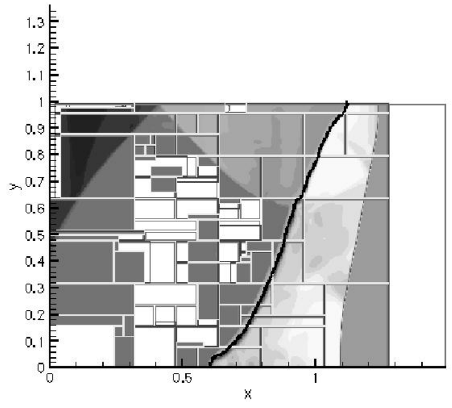

Figure 6. Density field at $\mathbf{t}/\tau = \mathbf{2.096}$, where t is the elapsed time and τ the time needed by the shock to traverse the oblique interface.

The Godunov method with **RK2** becomes unstable for strong shocks. The flexibility of CCA allows one to successfully reuse the code assembly in Fig. 5 to simulate strong shocks (Mach ≈ 3.5) by simply replacing the **GodunovFlux** component with **EFMFlux**, a component implementing a more diffusive gas-kinetic scheme [25].

5. Performance of Component-Assembled Code

In this section we examine the performance ramifications of the CCA architecture. All C and C++ codes were compiled using `gcc` (`g++`) version 2.95; Fortran libraries were compiled with Portland group's `pgf77`. `-O2` and `-fast` were used for the GNU and PG compilers respectively.

5.1. Single Processor Performance

This subsection addresses the question of serial performance of component-based codes. Typically the implemented method is called via a method in an interface, incurring the overhead of calling via a virtual function. This overhead is expected to be relatively small.

We created a code identical to the one in Sec. 4.1, except that the utilized mechanism had 8 species and 5 reactions (as opposed to 9 species and 19 reactions). The problem was solved on multiple identical cells, so that the elapsed time could be accurately measured with `getrusage()`. We deliberately used a light-weight RHS, so that the virtual function call would be a larger fraction of the computational time. Two tests were made by changing the number of times

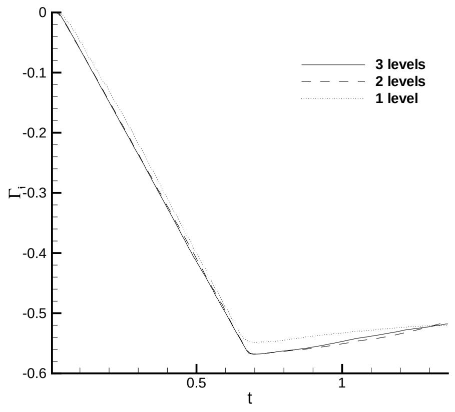

Figure 7. Convergence with respect to grid refinement of the interfacial circulation as the mesh hierarchy is allowed to have 1, 2 and 3 levels respectively.

the **ThermoChemistry** component was called (done by integrating for a longer time), in an effort to rack up larger overheads. The numbers are compared with those of a C-code in which the integrator (Cvode) was implemented as a library.

The codes were run on a 600 MHz AMD Athlon processor with 512 kB cache and 256 MB RAM, running RedHat 6.0. The results, in Table 4 indicate small differences in the performance with no clear trend. Δt refers to the time over which the problem was integrated, Ncells refers to the number of mesh points. NFE=number of RHS evaluations, *i.e* the number of times the **ThermoChemistry** component was called per cell. Timings are in seconds. % difference is the percentage difference in execution time. The overhead in CCA components is in the *method invocation*, not in the actual method execution. [11] shows how CCA method invocations are consistently ≈ 3 times more expensive than simple Fortran subroutine invocations; however since the invocation overhead itself is $O(10 - 100\text{ns})$, a more expensive component method invocation is still insignificant compared to the time spent in the method execution. Consequently, even for such a light-weight mechanism, the effect of the more expensive component overhead could not be reliably quantified. Thus, unless a design involves a very fine-grained decomposition, components do not adversely affect single processor performance.

5.2. Scaling To Multiple Processors

We ran the Reaction-Diffusion code (Sec. 4.3) on Sandia's CPlant cluster (433 MHz. EV56 Compaq Alpha processors, 192 MB RAM). Message passing was done using MPICH compiled for a 1 Gb/s Myrinet messaging fabric using 32-bit PCI32c cards. The code was run for 5 timesteps, each of 1×10^{-7}.

Δt	Ncells	NFE	Comp.	C-code	% diff.
1	1000	150	4.93	4.98	-0.88
1	5000	150	28.78	28.66	0.42
1	10000	150	58.71	58.19	0.89
10	1000	424	13.68	13.74	-0.44
10	5000	424	77.14	78.33	-1.54
10	10000	424	165.85	164.74	0.67

Table 4. Timings for the single-processor code. Comp. and diff. are abbreviations for Component and difference.

We performed constant single-processor and constant global problem size test to determine (a) communication-time characteristics as the machine size increases and (b) communication cost versus computational costs. Adaptivity was turned off since it renders scalability extremely sensitive to the performance of the load-balancer. Keeping the computational load on a processor fixed, the problem was run on up to 48 processors. Thus as the number of processors increased, so did the problem size. Each mesh point has 9 variables on it. Runs were done for single-processor domain sizes of 50×50, 100×100 and 175×175. In Fig. 8 we plot the run times for the three cases - we see that increasing the number of processors (and the problem size) does not make an appreciable difference. From Table 5 we see that the run times scale as the single-processor problem size. The problem size refers to the mesh size on each processor. The mean, median and standard deviations ($\bar{T}, \tilde{T}$ and σ respectively) for the data in Fig. 8 show that the machine behaves as "homogeneous", *i.e.* there are no sudden jumps in run-time as the job spreads itself across the machine. Timings are in seconds. Thus CPlant can be treated as a "homogeneous" machine, *i.e.* the communication times are not affected by machine size or the communication times are too small to be of any significance. We address this question next.

Problem Size	$\bar{T}$	$\tilde{T}$	σ
50×50	43.94	44.4	2.72
100×100	161.7	159.6	5.81
175×175	507.1	506.05	20.57

Table 5. Timings for the Reaction-Diffusion code.

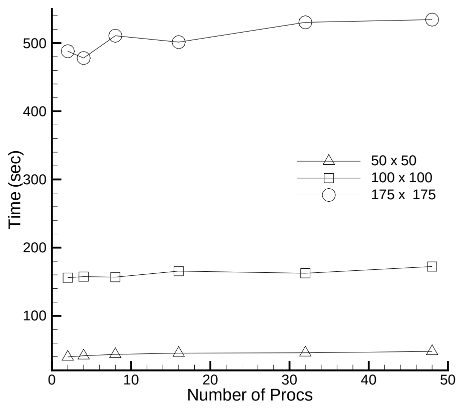

Figure 8. Timings for constant-processor-workload test for different single-CPU mesh sizes.

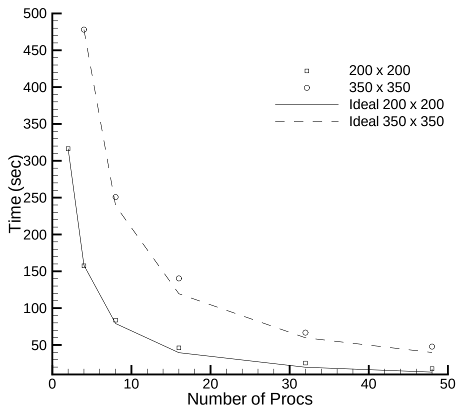

Figure 9. Scalability of the Reaction-Diffusion code (Fig. 2) without mesh refinement. The worst scalability efficiency is 73 % for the smaller problem on 48 nodes.

We ran two cases where the global problem size was kept constant while the number of processors were increased. It was expected that at a point the single-processor problem size would get small enough to be comparable to the communication costs. Fig. 9 shows the run time versus ideal run time for two problem sizes for up to 48 processors on Cplant [2]. The two problems treated are 200×200 and 350×350 on up to 48 nodes . The ideal curves are plotted in lines (solid for 200×200 and dashed for 350×350 meshes); individual measurements are symbols (boxes for 200×200 and circles for 350×350 meshes). We see that for the larger problem the measured run times follow the ideal closely. The parallel scaling efficiency $((t_1/P) * t_p$ where t_1 is the single-cpu run time and t_p is the run time on P processors) is found to be worst on a 200×200 grid on 48 processors - 73 %. The single-CPU problem size for this last run was 29×29.

6. Conclusions and Future Work

Computational science usually abstracts physical systems as systems of Partial or Ordinary Differential Equations. We have focused on a certain set of PDEs which are common in hydrodynamics and emphasized a certain class of solution methodologies (explicit integration of spatially coupled terms, implicit integration of all other terms). Within the boundaries set by these assumptions, we have developed a fairly general set of components that can be assembled to simulate very different kinds of flows. The CCA architecture is shown to be sufficiently flexible and simple to allow a straightforward design and development of components by computational scientists. The component architecture imposes negligible overhead vis-a-vis a traditional code and does not adversely affect parallel scalability.

Our study with CCA-based scientific components demonstrates the following:

1. Reuse of the **CvodeComponent** and **ThermoChemistry** components in the 0D ignition and the 2D Reaction-Diffusion simulations (instances CvodeComponent and ThermoChemistry in Fig. 1, instances CvodeSolver and ReactionTerms in Fig. 2).

2. Reuse of the **Mesh** and **ErrEstAndRegrid** components in the Reaction-Diffusion and shock-interface simulations (instances AMR_Mesh and ErrEstAndRegrid in Fig. 2, instances AMRMesh and ErrEstimator in Fig. 5).

3. Incorporation of a different numerical method in our 2D Shock-Interface Interaction, by replacing the **GodunovFlux** component with the **EFMFlux** component.

Recompilation/relinking of the code was not required. The components were developed within the group in a decoupled manner. Interfaces (and units) were agreed to and the components were coded to their agreed specifications. Language "interoperability" was achieved by wrapping C and Fortran 77 libraries into components; Decomposition of the code into subsystems was done first coarsely along numerical algorithm lines and then finely along physical models.

In the future, our thrust will be four-fold. (1) We will continue to develop numerical and physical components which will be used to simulate flames in SAMR. This will also include an effort to define interfaces to load-balancers prior to testing a number of them. (2) We will use CCA interoperability tools for the automated wrapping of C and Fortran libraries as components. (3) We will port our components to other implementations of the CCA standard, as parallel CCA frameworks become more interoperable. (4) By using TAU [19], we intend to characterize the performance characteristics of individual components and their assemblies.

Acknowledgements This work was supported by the Department of Energy, Office of Basic Energy Sciences, SciDAC Computational Chemistry Program. The authors would like to thank Kylene Smith for her contribution in the development of CvodeComponent.

References

[1] CORBA component model webpage. http://www.omg.com. Accessed July 2002.

[2] Cplant homepage. http://www.ca.sandia.gov/cacplant/.

[3] Grace homepage. http://www.caip.rutgers.edu/ parashar/TASSL/.

[4] SAMR homepage. http://www.cca-forum.org/∼jaray/documentation.html. Last accessed October 4, 2002.

[5] Visual basic webpage. http://msdn.microsoft.com/vbasic. Accessed July 2002.

[6] Xcat homepage. http://www.extreme.indiana.edu/ccat/. Also http://www.extreme.indiana.edu/xcat/; accessed July 2002.

[7] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl. The CCA core specifications in a distributed memory SPMD framework. *Concurrency: Practice and Experience*, 14:323–345, 2002. Also at http://www.cca-forum.org/ccafe03a/index.html.

[8] R. Armstrong, D. Gannon, A. Geist, K. Keahy, S. Kohn, L. McInnes, S. Parker and B. Smolenski. Towards a common component architecture for high performance scientific computing. In *Proceedings of the* 8^{th} *International Symposium on High Performance Distributed Computing*, 1999.

[9] B. P. Sommeijer, L. F. Shampine and J. G. Verwer. RKC: an explicit solver for parabolic PDEs. *J. Comp. Appl. Math.*, 88:315–326, 1998.

[10] M. J. Berger and P. Collela. Local adaptive mesh refinement for shock hydrodynamics. *J. Comp. Phys.*, 82:64–84, 1989.

[11] D. E. Bernholdt, W. R. Elwasif, J. A. Kohl, and T. G. W. Epperly. A component architecture for high-performance computing. In *Proceedings: Workshop in Performance Optimization via High-Level Languages and Libraries*, accepted.

[12] D. Brown, G. Chesshire, W. Henshaw and D. Quinlan. Overture : An object-oriented software system for solving partial differential equations in serial and parallel environments. In *Proceedings of the SIAM Parallel Conference*, Minneapolis, MN, USA, March 1997.

[13] S. D. Cohen and A. C. Hindmarsh. Cvode, a stiff/nonstiff ODE solver in C. *Computers in Physics*, 10(2):138–143, 1996.

[14] I. Crnkovic. Component-based software engineering – new challenges in software development. *Software Focus*, 2(4):127–133, 2002.

[15] J. D. de St. Germain, J. McCorquodale, S. G. Parker and C. R. Johnson. Uintah : A massively parallel problem solving environment. In *HPDC '00 : Ninth IEEE International Symposium on High Performance and Distributed Computing*, August 2000.

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

[17] C. Hirsch. *Numerical computation of internal and external flows, Volumes I and II*. John Wiley and Sons, 1988.

[18] S. Karmesin, J. Crotinger, J. Cummings, S. Haney, W. Humphrey, J.Reynders, S. Smith, and T. Williams. Array design and expression evaluation in POOMA II. In *Lecture Notes in Computer Science*, volume 1505. Springer, 1998.

[19] A. Malony and S. Shende. Performance technology for complex parallel and distributed systems. In G. Kotsis and P.Kacsuk, editors, *Distributed and Parallel Systems: From Concepts to Applications*, pages 37–46. Kluwer, Norwell, MA, 2000.

[20] H. Najm. A Conservative Low Mach Number Projection Method for Reacting Flow Modeling. In S. Chan, editor, *Transport Phenomena in Combustion*, volume 2, pages 921–932. Taylor and Francis, Wash. DC, 1996.

[21] H. N. Najm and P. S. Wyckoff and O. M. Knio A Semi-Implicit Numerical Scheme for Reacting Flow. I. Stiff Chemistry. *J. Comp. Phys.*, 143:381–402, 1998.

[22] B. Norris, S. Balay, S. Benson, L. Freitag, P. Hovland, L. McInnes and B. Smith. Parallel components for PDEs and optimization: Some issues and experiences. *Parallel Computing*, 28(12):1811–1831, 2002.

[23] M. Parashar and J. C. Browne. System engineering for high performance computing software: The HDDA/DAGH infratructure for implementation of parallel structured adaptive mesh refinement. In S. B. Baden, M. P. Chrisochoides, D. B. Gannon and M. L. Norman, editors, *Structured Adaptive Mesh Refinement*, volume 117 of *IMA*. Springer-Verlag, 2000.

[24] P. H. Paul. DRFM: A New Package for the Evaluation of Gas-Phase-Transport Properties. Sandia Report SAND98-8203, Sandia National Laboratories, Albuquerque, New Mexico, November 1997.

[25] D. I. Pullin. Direct simulation methods for compressible ideal gas flow. *J. Comp. Phys.*, 34:231–244, 1980.

[26] R. A. Yetter, F. L. Dryer and H. Rabitz. A comprehensive reaction mechanism for carbon monoxide/hydrogen/oxygen kinetics. *Combust. Sci. and Tech.*, 79:97–128, 1991.

[27] R. Samtaney, J. Ray and N. J. Zabusky. Baroclinic circulation generation on shock accelerated slow/fast gas interfaces. *Phys. Fluids*, 10(5):1217–1230, May 1998.

[28] G. Strang. On the construction and comparision of difference schemes. *SIAM J. Numer. Anal.*, 5:506–517, 1968.