Parallel Components for PDEs and Optimization: Some Issues and Experiences [1]

Boyana Norris, Satish Balay, Steven Benson, Lori Freitag,
Paul Hovland, Lois McInnes, and Barry Smith

Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Avenue, Argonne, IL 60439

Abstract

High-performance simulations in computational science often involve the combined software contributions of multidisciplinary teams of scientists, engineers, mathematicians, and computer scientists. One goal of component-based software engineering in large-scale scientific simulations is to help manage such complexity by enabling better interoperability among codes developed by different groups. This paper discusses recent work on building component interfaces and implementations in parallel numerical toolkits for mesh manipulations, discretization, linear algebra, and optimization. We consider several motivating applications involving partial differential equations and unconstrained minimization to demonstrate this approach and evaluate performance.

Key words: component-based software, scientific computing, parallel computing, interface specification
PACS: 89.90, 02.70, 07.05.T

Email addresses: `norris@mcs.anl.gov` (Boyana Norris), `balay@mcs.anl.gov` (Satish Balay), `benson@mcs.anl.gov` (Steven Benson), `freitag@mcs.anl.gov` (Lori Freitag), `hovland@mcs.anl.gov` (Paul Hovland), `mcinnes@mcs.anl.gov` (Lois McInnes), `bsmith@mcs.anl.gov` (Barry Smith).
[1] This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

1 Introduction

Computational scientists have benefited from the encapsulation of expertise in numerical libraries for many years. However, the complexity and scale of today's high-fidelity, multidisciplinary scientific simulations imply that development work often must be leveraged over many individual projects, because writing and maintaining a large custom application usually exceed the resources of a single group. These issues, along with the multilevel memory hierarchies of distributed-memory architectures, create ever more challenging demands for high-performance numerical software tools that are flexible, extensible, and interoperable with complementary research and industry technologies.

The Common Component Architecture (CCA) Forum [1,2], which includes researchers in various U.S. Department of Energy (DOE) laboratories and collaborating academic institutions, is developing a component architecture specification to address the unique challenges of high-performance scientific computing, with emphasis on scalable parallel computations that use possibly distributed resources. In addition to developing this specification, a reference framework, various components, and supplementary infrastructure, the CCA Forum is collaborating with practitioners in the high-performance computing community to design suites of domain-specific abstract component interface specifications.

This paper discusses initial experiences in developing CCA-compliant numerical component interfaces and implementations; see [1] for an introduction to the CCA approach and [3] for a discussion of related issues, including the design of a CCA-compliant framework. In this paper, we focus on interfaces at moderate granularities (for example, a linear solve or gradient evaluation) with support for multiple underlying implementations (for example, various mesh management techniques and algebraic solvers). In particular, we explore how well-defined interfaces in partial differential equation (PDE) solver and optimization components facilitate the use of external linear solver components that employ the Equation Solver Interface [4], under development by a multi-institution working group. We also explore the use of abstract interfaces for mesh management under development by the Terascale Simulation Tools and Technologies center [5]. In addition, we discuss experiences in adapting parts of existing parallel toolkits to function as CCA-compliant components that support these community-defined abstract interfaces.

The remainder of this paper motivates and explains our design strategy. Section 2 introduces several motivating applications, while Section 3 discusses recent work in numerical libraries and community-defined abstract interfaces. Section 4 presents background about component technologies for scientific computing, including the approach under development by the CCA Forum. In Section 5, we present newly developed components for mesh management, discretization, linear algebra, and optimization, and in Section 6 we evaluate component reuse and performance in

several applications. Section 7 discusses observations and directions for future work. Throughout the paper, we adhere to some simple conventions for differentiating between interfaces and implementations: abstract interfaces are *italicized*, while components and other concrete implementations use a `fixed-width font`.

2 Motivating Simulations

The complexity of large-scale scientific simulations often necessitates the combined use of multiple software packages developed by different groups. For example, several multidisciplinary projects that motivate this work involve computational astrophysics [6], chemistry [7,8], and fusion [9]; each has challenging resolution and complexity requirements that demand massively parallel computing resources and a range of sophisticated software. Such applications typically involve areas such as discretization, partitioning, load balancing, adaptive mesh manipulations, scalable algebraic solvers, optimization, parallel data redistribution, parallel input/output, performance diagnostics, computational steering, and visualization. Moreover, the state of the art in each of these areas is constantly evolving, necessitating frequent software updates during the lifetime of a given application.

To assist in explaining the design and evaluating the performance of newly developed prototype high-performance numerical components that are the focus of this paper, we consider three motivating applications: a steady-state PDE, a time-dependent PDE, and an unconstrained minimization problem. We have deliberately chosen these examples to be relatively simple and therefore straightforward to explain, yet they incorporate numerical kernels and phases of solution that commonly arise in the more complicated scientific simulations that motivate our work.

Steady-State PDE Application. The first example is a simple application that solves $\nabla^2\phi(x,y) = 0, \quad x \in [0,1], \quad y \in [0,1]$, with $\phi(0,y) = 0$, $\phi(1,y) = \sin(2\pi y)$, and $\frac{\partial\phi}{\partial y}(x,0) = \frac{\partial\phi}{\partial y}(x,1) = 0$, using mesh, discretization, and linear solver components. We employ a linear finite element discretization and an unstructured triangular mesh generated using the Triangle package [10]. This example has characteristics of the large, sparse, linear systems that are at the heart of many scientific simulations, yet it is sufficiently compact to enable the demonstration of CCA concepts and code in Section 4.2.

Time-Dependent PDE Application. The second PDE that we consider is the heat equation, given by $\frac{\partial\phi}{\partial t} = \nabla^2\phi(x,y,t), \quad x \in [0,1], \quad y \in [0,1]$, with $\phi(0,y,t) = 0$, $\phi(1,y,t) = \frac{1}{2}\sin(2\pi y)\cos(t/2)$, $\frac{\partial\phi}{\partial y}(x,0,t) = \frac{\partial\phi}{\partial y}(x,1,t) = 0$. The initial condition is $\phi(x,y,0) = \sin(\frac{1}{2}\pi x)\sin(2\pi y)$. As discussed in Section 6.1, this application reuses the mesh, discretization, and linear algebra components employed by the steady-state PDE example and introduces a time integration component.

Unconstrained Minimization Application. We also consider an unconstrained minimization example taken from the MINPACK-2 test suite [11]. Given a rectangular two-dimensional domain and boundary values along the edges of the domain, the objective is to find the surface with minimal area that satisfies the boundary conditions, that is, to compute $min\ f(x)$, where $f : \Re^n \longrightarrow \Re$. This example, which is analogous in form to several computational chemistry applications [7,8] that motivate this work, reuses the linear algebra components employed for the PDE examples and introduces a component for unconstrained minimization.

3 Parallel Numerical Libraries and Common Interface Efforts

As the complexity of computational science applications has increased, the use of object-oriented software methods for the development of both applications and numerical toolkits has also increased. The migration toward this approach can be attributed in part to the encapsulation and reusability provided by well-designed objects, which enable developers to focus on a small part of a complex system rather than attempting to develop and maintain a monolithic application. Furthermore, reuse justifies expending significant effort on the development of highly optimized toolkits encapsulating expert knowledge, such as Diffpack [12], Overture [13], ParPre [14], SAMRAI [15], and PETSc [16,17].

As previously discussed, the applications of interest within high-performance scientific computing often require the combined use of software tools that encapsulate the expertise of multidisciplinary teams. Current-generation tools have demonstrated good success in pairwise coupling, whereby one tool directly calls another by using well-defined interfaces that are known at compile time. Our earlier work on building two-way interfaces between SUMAA3d [18] and PETSc (discussed in [19]), between Overture [13] and PETSc (discussed in [20]), and between PVODE [21] and PETSc (discussed in [22]), showed that interfacing two sophisticated numerical software tools typically requires an in-depth understanding of each tool's interface and implementation. Since developing these interfaces is often labor-intensive, experimentation with tools providing alternative technologies is severely inhibited. For example, the left-hand portion of Figure 1 shows the current situation in which $m \times n$ individual interfaces are needed to experiment with different combinations of mesh management infrastructures, such as Distributed Arrays [17], Overture [13], PAOMD [23], and SUMAA3d [18], and algebraic solvers in packages such as ISIS++ [24], PETSc [17], and Trilinos [25].

Common interfaces enable users to leverage expertise encapsulated within various underlying implementations without needing to commit to a particular solution strategy and to risk making premature choices of data structures and algorithms. To enable this flexibility, various communities are beginning to define domain-specific suites of common interfaces. Once such interfaces have been developed, application

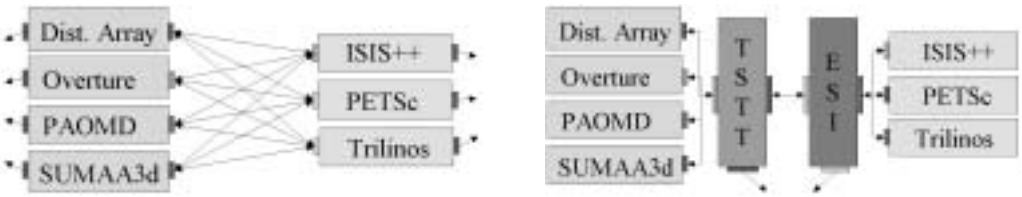


Fig. 1. (*Left*): The current interface situation connecting m mesh management systems to n linear system solvers through $m \times n$ interfaces. (*Right*): The desired interface situation, in which many tools that provide similar functionality are compliant with a single interface.

scientists may employ them in their codes and then easily explore the use of any compliant toolkit. This approach reduces the interface development effort needed to experiment with a number of software tools, as is illustrated in the right-hand portion of Figure 1. In this scenario, the mesh management infrastructures and linear solvers would each write to abstract interfaces defined by their respective communities. These interfaces then would serve as the point of entry to many different underlying implementations for tool-to-tool and tool-to-application interactions.

We note that interface definition efforts are complex and time-consuming endeavors, which are complicated by the need to provide a rich set of functionalities while preserving the commonality of the interfaces. Additional challenges include the need to maintain high efficiency and to support a variety of scientific programming languages. Current research in creating common interface specifications for numerical tools includes efforts in the algebraic solver community through the Equation Solver Interface (ESI) working group and the mesh management community through the Terascale Simulation Tools and Technologies (TSTT) SciDAC center.

3.1 The Equation Solver Interface

One of the most computationally intensive phases that arises in many scientific applications is the solution of discretized linear systems of the form $Ax = b$. Preconditioned Krylov methods are effective for the parallel solution of such problems, although algorithmic performance varies considerably depending on the underlying physics being modeled. Scientists could benefit from the ability to experiment more easily with the variety of preconditioners and Krylov methods provided in different parallel linear solver libraries without having to perform the labor-intensive task of manually writing a different interface from application code to each linear algebra toolkit. Such flexibility would enable applications to incorporate new algorithms with better latency tolerance or more efficient cache utilization as these are discovered and encapsulated within toolkits.

The Equation Solver Interface Forum [4] is one effort addressing these issues; other related work includes [26]. The ESI working group, which was formed in 1997 by researchers in various DOE laboratories, continues to meet regularly and welcomes

participation from the scientific community. The ESI specification defines abstract interfaces for manipulating objects that are commonly used in the scalable solution of linear systems. For example, the *Operator* interface supports matrices, preconditioners, and solvers by viewing them as linear operators. The *Operator* interface defines the `setup` method for initializing the operator and the `apply` method for applying the operator to an ESI *Vector* and storing the result in another ESI *Vector*. The base ESI class *Object*, from which all other interfaces are subclassed, contains mechanisms for reference counting and specifying supported interfaces.

3.2 The TSTT Mesh Interface

Just as many different algebraic solvers provide similar functionality, many tools are available that generate a variety of mesh types, ranging from unstructured meshes to overlapping structured meshes and hybrid meshes [27]. Approximation techniques used on these meshes include finite difference, finite volume, finite element, spectral element, and discontinuous Galerkin methods. Various combinations of these mesh and approximation types may be used to solve PDE-based problems. The fundamental concepts are the same for all approaches: some discrete representation of the geometry (the mesh) is used to approximate the physical domain, and some discretization procedure is used to represent approximate solutions and differential operators on the mesh. In addition, the concepts of adaptive mesh refinement, time-varying meshes, data transfer between different meshes, and parallel mesh decomposition are the same regardless of their implementations. The software tools providing these capabilities are becoming increasingly accepted by the scientific community, but their application interfaces are incompatible. Common interface specification would enable significantly more experimentation by application scientists to determine which discretization strategy most accurately and efficiently captures the physical phenomenon of interest.

To facilitate the development of such interfaces, the Department of Energy has recently funded the Terascale Simulation Tools and Technologies center [5], which has a pervading theme of developing interoperable and interchangeable meshing and discretization software. Current emphasis is on creating common interfaces for querying existing TSTT mesh management technologies that will allow them to interoperate with each other. The interfaces focus on access to information pertaining to low-level mesh objects such as vertices, edges, faces, and regions. A small set of interfaces for accessing spatial (e.g., vertex coordinates) and topological (e.g., adjacency information) is also being developed. Discussions are under way to determine interfaces for mesh services, canonical ordering of entities, and query interfaces for distributed meshes in a parallel computing environment.

4 High-Performance Component Technologies

Even with well-designed libraries and suites of standard interfaces, sharing object-oriented code developed by different groups is difficult because of language incompatibilities, the lack of standardization for interobject communication, and the need for compile-time coupling of interfaces. Component-based software design combines object-oriented design with the powerful features of well-defined interfaces, programming language interoperability, and dynamic composability [28]. While component-based design was initially motivated by the needs of business application developers, it also offers enormous potential benefits to the computational science community by encouraging the development of interoperable software and the dynamic construction of new algorithms and applications.

4.1 Overview of the Common Component Architecture

The most popular frameworks for component-based software engineering, namely Microsoft's Component Object Model (COM) [29] and Sun's (Enterprise) JavaBeans (EJB) [30,31], are suitable for some computational science applications [32]. As discussed in [1,22], however, many large-scale scientific applications require features not provided by these industry component technologies. For example, these frameworks were not designed to enable tight coupling of components executing on massively parallel machines that may be networked in a distributed environment and hence do not address issues of collective connections and parallel data redistribution among distributed components. Furthermore, some of these frameworks are either architecture-specific (e.g., COM), or language-specific (e.g., EJB). Finally, industry component approaches do not support interoperability with important scientific languages, such as Fortran 90, or lightweight connections between parallel components. To address this need, the Common Component Architecture Forum [1,2] is developing a component model specification that provides the features required by advanced computational science applications while remaining compatible with COM, EJB, and the CORBA component architecture [33,34] to the fullest extent possible. The CCA model defines a very lightweight mechanism for interactions between massively parallel components and supports interoperability between languages widely used in scientific applications. This effort builds on the experience of research groups who study high-performance component architectures and related design issues, including [35–42]. Other related work in high-performance scientific computing includes [43–46].

The current CCA specification consists of (1) a core portion defining an interface that a component must implement to connect to another component and (2) a collection of public interfaces, or **ports** [1,47]; these concepts are demonstrated in an example in Section 4.2. Ports can define the interactions between relatively tightly

coupled parallel numerical components, which typically require very fast communication for scalable performance; ports can also define loosely coupled interactions with possibly remote components for monitoring, analysis, and visualization. CCA ports employ a **provides/uses** paradigm, whereby a component provides a set of interfaces that other components can use; Section 5.1 presents some examples. A **provides** port is essentially a set of functions that are executed by the component on behalf of the component's "users". An external "builder" tool, which may also be implemented as a CCA component, connects the **provides** ports of one component to the **uses** ports of another. The **uses** ports of a component can be viewed as the connection points available to other components as well as the framework, where we consider a CCA framework to be a software environment that enables dynamic instantiation, coupling, and method invocation on components. The simulations in this work employ the CCAFFEINE framework [3], which is further discussed in Section 4.3. In the remainder of this paper, we also refer to ports as interfaces or abstract interfaces, since ports are abstract by definition.

Another key facet of the CCA approach is the development of an interface language called SIDL (Scientific Interface Definition Language) [35,48], which provides language interoperability for CCA ports. Since scientific applications often require the integration of components written in a variety of languages, such as Fortran, C, C++, Python, and Java, support for language interoperability is critical. While the prototype component applications presented in this work all use port interfaces written in C++, we plan to incorporate SIDL interfaces in future versions.

4.2 CCA Components

A software component is an encapsulated software object that provides a certain set of functionalities or services and can be used in conjunction with other components to build applications. In general, a component consists of one or more abstract interfaces and one or more implementations, and conforms to a prescribed behavior within a given computational framework. The CCA component specification defines a set of rules for implementing components and for the behavior components must exhibit to coexist with other components in a CCA-compliant framework. In particular, to be CCA-compliant, a C++ component class must

- inherit from the abstract *Component* interface,
- contain a private data member of type `Services*`, which is a handle to the `Services` object provided by a CCA-compliant framework, and
- declare a public function `setServices(Services *cc)`, which is used by the framework to set the `Services` handle.

To illustrate these requirements in more detail, we consider the driver component for the steady-state PDE application described in Section 2. This compo-

nent uses mesh, discretization, solver, and visualization components and provides a *gov::cca::GoPort*, a standard CCA port that provides an entry point to the application, similar to a traditional `main` routine in Fortran or C. The class definition file for the driver component is shown in Figure 2. Two include files are needed: `cca.h` and `stdPorts.h`, which contain the CCA specification and several standard ports such as the *GoPort*, respectively. The `DriverComponent` class inherits from *gov::cca::Component* and *gov::cca::GoPort* and implements their public methods `setServices` and `go`, respectively. In addition, the DriverComponent class has a private data member of type `gov::cca::Services`.

```
#include <cca.h>
#include <stdPorts.h>

class DriverComponent :  public virtual gov::cca::Component,
                         public virtual gov::cca::GoPort {

private:
  gov::cca::Services *svc;

public:
  DriverComponent();
  virtual ~DriverComponent();
  virtual void setServices(gov::cca::Services *cc);
  virtual int go();
};
```

Fig. 2. The class definition file for the steady-state driver component.

When the component is instantiated, the framework accepts a pointer to the component and holds the pointer for the lifetime of the component. Other components obtain and release the component's ports only through requests made to the framework via the `gov::cca::Services` object. This object is passed to the `setServices` method when each component is created; typical usage for the `DriverComponent` example is shown in Figure 3. For each *uses* port, two methods on the `gov::cca::Services` object are called to register the request with the framework. First, a `PortInfo` object containing a string identifier name (e.g., `mesh`) and the abstract port type (e.g., *MeshPort*) is created with the call to `createPortInfo`. The `PortInfo` object is then passed to the framework via the `registerUsesPort` method. The framework keeps a record of all such requests to ensure that ports are properly matched when components are created and connected.

Although they are not shown in Figure 3, similar calls are made for the discretization, solver, and visualization ports needed by the driver. For each port that the driver provides, in this case a *GoPort*, a similar protocol is used. Again, a `PortInfo` object is created using `createPortInfo` and passed to the framework with the `addProvidesPort` method. If the component is to be shut down, the framework passes in a `NULL` services object. In this case the driver component unregisters and releases any **uses** ports and removes framework access to its **provides** ports.

```
void DriverComponent::setServices(gov::cca::Services *cc)
{
  gov::cca::Port *p;

  if (cc == 0) {    // Close down if not closed already
    svc->unregisterUsesPort("MeshPort");
    svc->releasePort("mesh");
    svc->removeProvidesPort("go");
    return;
  }

  // register the uses and provides ports
  svc->registerUsesPort(svc->createPortInfo("mesh","MeshPort",0));
  svc->addProvidesPort(p, svc->createPortInfo("go","gov.cca.GoPort",0));
}
```

Fig. 3. The `setServices` method for the steady-state driver component.

```
  MeshPort *mesh_ptr;

  gov::cca::Port *p = svc->getPort("mesh");
  mesh_ptr = dynamic_cast< MeshPort * >(p);
  ...

  mesh_ptr->GetVertices();
  ...

  svc->releasePort("mesh");
```

Fig. 4. The methods needed to access the `MeshPort` in the driver component.

External ports are accessed through the `Services` object method `getPort`, as illustrated in Figure 4 for *MeshPort*. The generic port returned from `getPort` is dynamically cast to the *MeshPort* type and assigned to `mesh_ptr` for use as illustrated with the `mesh_ptr->GetVertices` method. When the port is no longer needed, it can be released by using the `gov::cca::Services releasePort` method. If *MeshPort* functionality is needed later, it can be reacquired from the same component or from a different component that also provides *MeshPort*.

4.3 CCA Frameworks

Until recently, the draft CCA specification did not address framework construction or the mechanisms used for instantiating and connecting components. As a result, several different frameworks suited for different situations have been implemented [3,36,37]. Some optimize the use of components distributed across a wide-area Grid, while others target massively parallel Single Program Multiple Data (SPMD) components. The components described in this paper were tested in the CCAFFEINE environment [3], which supports SPMD-style computing.

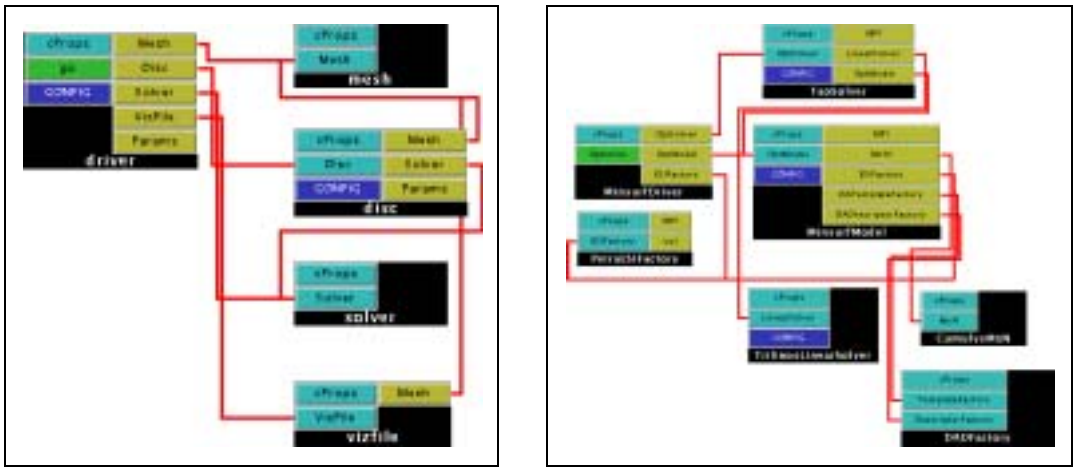
The CCAFFEINE concept of SPMD component computing is a simple generalization of the traditional SPMD approach. In conventional SPMD computing, there is

one program per process. In the CCAFFEINE implementation of SPMD component computing, each instance of a component class is created on all the processes on which the framework instance is running. The concept is similar to creating objects in an MPI environment, with the restriction that the objects comply with the CCA interface standards. In a single CCAFFEINE framework instantiation, components are connected in the same process address space only. This does not preclude interactions with remote components, but such interactions are outside of the framework's control.

Despite the varying environments in which different framework implementations operate, the high-level functionality provided by any given framework is largely the same. The CCA Forum has recently adopted several framework interfaces that abstract the implementation of component management mechanisms without limiting their functionality. These interfaces allow a programmer to create applications by manipulating a CCA-compliant framework via standard interfaces for component discovery, instantiation, connection, and destruction.

5 Numerical Scientific Components

In this section we describe the abstract interfaces developed while creating the components for the applications introduced in Section 2. We also discuss the components implementing these interfaces and briefly describe the underlying noncomponent software used in the implementations.



(a) Steady-state PDE (b) Unconstrained minimization

Fig. 5. The component wiring diagrams for (a) the steady-state PDE example, and (b) the unconstrained minimization example.

Figure 5 illustrates the composition of two of the applications using the graphical interface of the CCAFFEINE framework. The black boxes in the wiring diagram

for the steady-state PDE example show the mesh, discretization, linear solver, visualization, and driver components (see Figure 5(a)); the lines represent connections between **uses** and **provides** ports, which are gold and blue, respectively. For example, the discretization component's *Mesh* **uses** port is connected to the mesh component's *Mesh* **provides** port; hence, the discretization component can invoke the *TSTTMeshQuery* interface methods that the mesh component has implemented. The special *GoPort* (named "go" in this application) is used to start the execution of the application. Figure 5(b) shows the components involved in the solution of an unconstrained minimization problem. The optimization solver component in this snapshot has been configured to use an inexact Newton method, which requires the solution of a linear system. The diagram also includes components for parallel data description and redistribution.

5.1 Component Interfaces and Implementations

Some of the community-defined interfaces described in Section 3, for instance, *TSTTMeshQuery*, directly correspond to the CCA *ports* of the component implementations of the applications in Section 2. The ESI common interface specification is used in two different ways: as a mechanism for passing vectors and matrices between components, and as an abstract interface for linear solver components. However, not all functionality required for the component implementations of these applications was available in the form of commonly accepted abstract interfaces. We describe the new interfaces used in the applications in Section 2, the components implementing these interfaces, and the approximate level of development effort. The interfaces (italicized) and their corresponding implementations (with class names in a fixed-width font) are discussed below.

- *ESIFactory.* The ESI specification contains no provisions for object instantiation. Thus, we have defined an *ESIFactory* port, which is an abstract factory interface for instantiating objects from an ESI interface implementation.
  Implementation: The `ESIFactory_Petra` component supports the creation of ESI-compliant index spaces, vectors, matrices, and solvers. The underlying software is Trilinos (more specifically, the Epetra library) [25]. The `ESIFactory_Petsc` component supports the creation of ESI-compliant index spaces, vectors, and matrices using PETSc [16] as the underlying software. The main development effort was adding ESI support to PETSc and portions of Trilinos. The factory components' implementations are thin wrappers over methods available in each underlying toolkit and thus took little programming effort.
- *LinearSolver.* The *LinearSolver* port is derived from the ESI *Solver* interface. The objective is to enable linear solver components to provide extended functionality that is not part of the ESI specification. Currently, the only additional methods provided are `initialize` and `finalize`.
  Implementation: The `LinearSolver_Trilinos` component provides a *Lin-*

earSolver port and is based on the AztecOO library, which is part of the Trilinos project [25]. The `LinearSolver_Petsc` component implementation is based on the PETSc library [16] and specifically uses the Scalable Linear Equations Solver (SLES) interface to various Krylov methods and preconditioners. Again, the principal development effort in implementing these components focused on ESI support within the original toolkits. The implementation of component-specific functionality was straightforward.

- *TSTTMeshQuery*. Using the TSTT interfaces mentioned in Section 3.2, we developed a mesh component for a static, unstructured triangular grid.
  Implementation: The `TSTTMesh` component provides access to node and element information (edge and face data will be supported shortly) and was sufficient to implement linear, finite-element discretization for the diffusion PDE operators introduced in Section 2. The primary development effort was the separation of the mesh from the remainder of the application. Once this had been accomplished, the mesh query interface and component-specific functionality were straightforward to implement.
- *FEMDiscretization*. This interface provides linear, finite-element discretizations for commonly used PDE operators and boundary conditions. It currently works for unstructured triangular meshes accessed through the TSTT interfaces mentioned in Section 3.2. It provides the matrix and vector assembly routines to create the linear systems of equations necessary to solve the steady-state and time-dependent PDE applications introduced in Section 2.
  Implementation: The discretization component provides approximations for advection and diffusion operators as well as Dirichlet and Neumann boundary conditions with either exact or Gaussian quadrature. The interface developed here is specific to this component and was therefore straightforward to implement. This interface is expected to evolve as the TSTT discretization library is developed. This component uses the *TSTTMeshQuery* and *LinearSolver* ports.
- *OptimizationSolver.* The *OptimizationSolver* port defines a prototype high-level interface to optimization solvers, which closely mirrors the TAO optimization solver interface; this interface is expected to evolve as common interface definition efforts for optimization software progress.
  Implementation: The `TaoSolver` component implementation is based on the Toolkit for Advanced Optimization (TAO) [49], which provides a growing number of algorithms for constrained and unconstrained optimization. The single abstract interface for the *TaoSolver* component enables the user to employ a variety of solution techniques for the unconstrained minimization problem introduced in Section 2, including Newton-based line search and trust region strategies, a limited-memory variable metric method, and a nonlinear conjugate gradient method. The implementation is a thin wrapper over existing TAO interfaces, and thus took minimal effort to develop.
- *OptimizationModel.* The *OptimizationModel* port includes methods that define the optimization problem and inherits from the abstract TAO *ESIApplication* interface. The *OptimizationModel* interface includes methods for function, gradient, Hessian, and constraint evaluation.

Implementation: The `MinsurfModel` component implements the minimum surface area model described in Section 2. The development of this component required moderate effort and involved a conversion of an existing TAO example to use the ESI for vectors and matrices and to implement the component-specific functionality. Many different optimization problems can be implemented by providing the *OptimizationModel* port, which is then used by solver components such as `TaoSolver`.

While developing a high-level CCA component interface on top of an existing numerical library is relatively straightforward, the design issues are typically much more complex when one aims to provide interoperability between components developed by different groups. We have found that the amount of effort needed to develop interoperable numerical components on top of an existing library depends on the design of the underlying software, including the degree to which abstractions and encapsulation have already been employed in the library's user interface and internal design. For example, a significant part of the overall development of the optimization component `TaoSolver` involved incorporating community-defined abstract interfaces for linear algebra within the existing TAO library so that the components implementing the *LinearSolver* interface could be used to solve linear subproblems arising in the optimization algorithms. TAO itself, however, required no other changes because its design already incorporated abstractions for vectors, matrices, and linear solvers. In contrast, if starting with underlying software that does not already employ abstractions and encapsulation, more effort would be required both to build and to use interoperable components.

Other Interfaces

We mention additional interfaces and implementations that are used in these applications and have been developed by collaborators within the CCA Forum. Further information about these interfaces, as well as those presented above, is available at `http://www.cca-forum.org/cca-sc01`.

- ODEPACK++. The time integration routines are provided by the ODEPACK++ component, which was developed by Ben Allan of Sandia National Laboratories and is based on the ODEPACK library [50].
- Parallel Data Redistribution. To redistribute data between various application and visualization components, we employ the "MxN" parallel data redistribution specification that is under development by a CCA working group. This interface supports the connection and data transfer between two parallel components that may be distributed across different numbers of processes. Our examples employ a CUMULVS-based [42] component developed by Jim Kohl of Oak Ridge National Laboratory (ORNL) for the "Mx1" transfer of data between application and visualization components.
- Distributed Array Descriptors. To describe the layout of data over multiple processes and in local memory so that we can perform "MxN"-style parallel data

redistribution, we use distributed array descriptor interfaces, which have been developed by David Bernholdt (ORNL) as part of activities within a CCA working group on scientific data components.

- Visualization. These applications can employ a simple *VizFile* port to print data associated with the vertices of a TSTT mesh in MATLAB or vtk file formats. We can also employ visualization components developed by Jim Kohl of ORNL, which use the "MxN" and distributed array descriptor interfaces and are based on CUMULVS [42].

5.2 Underlying Software

The parallel numerical software underlying these components includes linear algebra capabilities within PETSc [16,17] and Trilinos [25], optimization software within TAO [51,49], and meshing technology within the TSTT [5].

One benefit of the *ESIFactory* and *LinearSolver* interfaces introduced in Section 5.1 is that we can readily incorporate various underlying libraries that support ESI interfaces. In particular, `ESIFactory_Petsc` and `LinearSolver_Petsc` use newly developed ESI interfaces to vectors, matrices, and linear solvers within PETSc [16,17], a suite of software for the scalable solution PDE-based applications that integrates a hierarchy of code ranging from low-level distributed data structures for parallel vectors and matrices through high-level linear, nonlinear, and timestepping solvers. Likewise, `ESIFactory_Petra` and `LinearSolver_Trilinos` employ new ESI-compliant interfaces to vectors, matrices, and linear solvers within Trilinos [25], a set of parallel solver libraries for the solution of large-scale, multiphysics scientific applications. Thus, component-based applications, including all three examples introduced in Section 2, can use a single set of abstract interfaces to experiment easily with a broad range of Krylov methods and preconditioners that are currently provided in PETSc and Trilinos; moreover, future algorithms and toolkits that support these interfaces will also be accessible without having to modify application-specific components.

The `TaoSolver` component employs the *ESIFactory* and *LinearSolver* interfaces and builds on the Toolkit for Advanced Optimization (TAO) [51,49], which focuses on scalable optimization software, including nonlinear least squares, unconstrained minimization, bound-constrained optimization, and general nonlinear optimization. TAO optimization algorithms use high-level abstractions for matrices, vectors, and linear solvers and can employ various external software tools for these capabilities. The primary parts of TAO used in this work are parallel unconstrained minimization solvers, including new support for ESI interfaces.

6 Results

We first discuss component reuse among the three motivating applications and then evaluate component performance.

6.1 Component Reuse

All three component implementations of the applications discussed in Section 2 reuse some subset of the components described in Section 5.1. Reuse was achieved both through community-defined interfaces and through ports developed specifically for use with these applications.

Both PDE-based applications reuse components that implement the *TSTTMeshQuery* and *FEMDiscretization* interfaces for mesh management and discretization as well as the *ESIFactory* and *LinearSolver* interfaces for linear algebra (see Figure 5(a) for the steady-state case). The time-dependent application also employs the ODEPACK++ components for time integration and visualization components for real-time access to the solution field. The data are transfered from running simulations to visualization environments using the `CumulvsMxN` data transfer component that employs the `DistArrayDescriptorFactory` for data description.

Among the components involved in the unconstrained minimization problem, those used in other applications include the PETSc and Trilinos implementations of the *ESIFactory* and *LinearSolver* interfaces. Further, as for the PDE-based examples, the `DistArrayDescriptorFactory`, `CumulvsMxN`, and `VizProxy` implementations can be used to describe, redistribute, and visualize the current solution. The components specific to this application include `MinsurfModel`, which implements the *OptimizationModel* interface, and the TAO-based implementation of the *OptimizationSolver* interface (see Figure 5(b)).

6.2 Component Performance

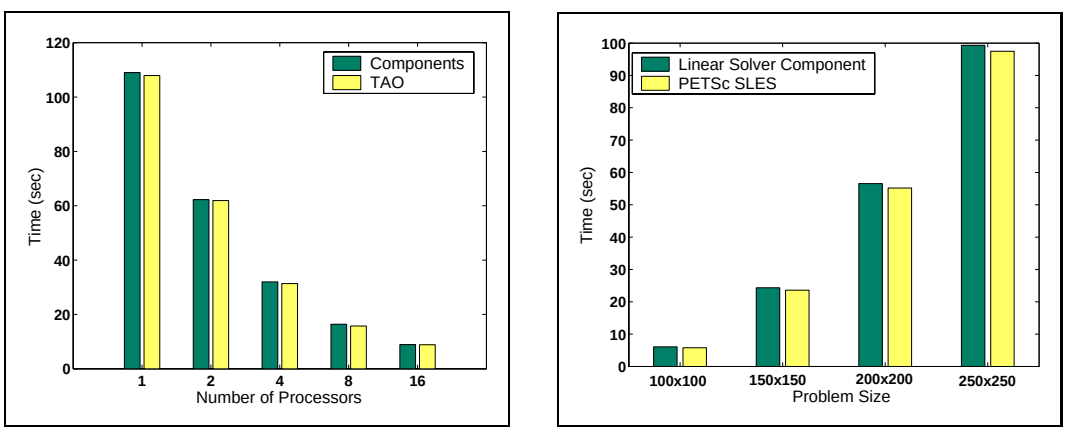
One of the main goals of the CCA specification is to achieve high performance in parallel code; however, a common concern about the use of CCA components is the effect that component overhead may have on performance. We conducted experiments to evaluate the performance differences between the component and library-based implementations of the minimum surface optimization problem introduced in Section 2. Our parallel results were obtained on a Linux cluster of dual 550 MHz Pentium-III nodes with 1 GB of RAM each, connected via Myrinet. The uniprocessor results were obtained on a 400 MHz Pentium-III Linux workstation with 256 MB of RAM.

These particular experiments used an inexact Newton method with a line search, which has the following general form for an n-dimensional unconstrained minimization problem:

$$x_{k+1} = x_k - \alpha[\nabla^2 f(x_k)]^{-1}\nabla f(x_k), \quad k = 0, 1, \ldots,$$

where $x_0 \in \Re^n$ is an initial approximation to the solution, and $\nabla^2 f(x_k)$ is positive definite. Newton-based methods (see, e.g., [52]) have proven effective for many large-scale problems, as they offer the advantage of rapid convergence when an iterate is near to a problem's solution, and line search techniques can extend the radius of convergence.

Each iteration of the Newton method requires a function, gradient, and Hessian evaluation, as well as an approximate solution of a linear system of equations to determine a step direction. The component wiring diagram in Figure 5(b) illustrates these interactions via port connections; the library-based version of code is organized similarly, although all software interactions occur via traditional routine calls within application and library code instead of employing component ports. We solved the linearized Newton systems approximately with a variety of preconditioned Krylov methods; the parallel results presented in Figure 6 used the conjugate gradient method and block Jacobi preconditioner with no-fill incomplete factorization as the solver for each subdomain, while the uniprocessor results used the conjugate gradient method with a no-fill incomplete factorization preconditioner.



(a) Total execution time

(b) Cumulative linear solver time

Fig. 6. Component overhead for (a) finding the solution of the unconstrained minimization on a 250 $\times$ 250 grid and (b) the `LinearSolver_Petsc` component.

By using abstract interfaces at several levels of the implementation, the component version introduces a number of virtual function calls, including matrix and vector access operations, linear solver methods, and function, gradient, and Hessian evaluation routines used by the Newton solver. Figure 6(a) depicts the difference in total execution time between the component implementation and the original application

on 1, 2, 4, 8, and 16 processors for a fixed-size problem of dimension 250×250. Overall, the virtual function call overhead is negligible, with the most significant performance penalty occurring for small problem sizes. This figure also illustrates that good scaling behavior is not lost by the use of CCA components: the time for the total component-based computation on one processor is 109 seconds, while the corresponding time using a sixteen-processor linux cluster is 9 seconds.

Figure 6(b) illustrates the performance of the CCA-related code in the `LinearSolver_Petsc` component for various problem sizes on a single processor. As the problem size increases, this fixed overhead becomes less significant, accounting for 2 to 5 percent of the total linear solution time. This overhead consists of the cost of the virtual function calls resulting from the use of CCA ports and abstract common interfaces (e.g., ESI) for data exchange between components.

7 Conclusions

By exploring several motivating scientific applications, we have presented newly developed high-performance components for discretization, mesh management, linear algebra, and optimization that are compliant with the emerging CCA specification. The complete source code and documentation for these components and applications are available via the Web site `http://www.cca-forum.org/cca-sc01` as part of a distribution of tutorial-style CCA codes.

We have demonstrated that the CCA approach to component-based design enables the integration of high-performance numerical software tools developed by different groups. In particular, the direct-connect variant of the CCA **provides/uses** ports interface exchange mechanism enables connections that do not impede intercomponent performance in tightly coupled parallel computations within the same address space, such as the interaction between components for unconstrained minimization and linear system solution. Well-defined component ports provide a clear separation between abstract interfaces and underlying implementations, and dynamic composability facilitates experimentation among different algorithms and data structures.

Future research will include ongoing collaborations to explore the further use of numerical components in large-scale scientific simulations and to develop domain-specific abstract interface specifications in conjunction with other researchers. We will also explore quality-of-service issues for numerical components, including how to determine suitable matches between the requirements of **user** components (e.g., a minimizer) for accuracy, robustness, performance, and scalability and the capabilities of various component **provider** implementations (e.g., a linear solver). In addition, other researchers are developing tools to help automate the process of creating CCA-compliant components from legacy codes [53].

Acknowledgments

We have developed the ideas presented in this paper from research and discussion with collaborators within the Common Component Architecture (CCA) Forum [2] and the Terascale Simulation Tools and Technologies Center [5]. We thank these groups for stimulating discussions of issues in component and interface design for high-performance scientific computing. We thank Alan Williams (SNL) for implementing the ESI interface for Epetra objects and AztecOO solvers. We also thank Gail Pieper for proofreading a draft manuscript.

References

[1] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. C. McInnes, S. Parker, B. Smolinski, Toward a common component architecture for high-performance scientific computing, in: Proceedings of High Performance Distributed Computing, 1999, pp. 115–124.

[2] Common Component Architecture Forum, see `www.cca-forum.org`.

[3] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, J. A. Kohl, The CCA core specification in a distributed memory SPMD framework, Concurrency and Computation: Practice and Experience (to appear).

[4] Equation Solver Interface Forum, see `z.ca.sandia.gov/esi`.

[5] Terascale Simulation Tools and Technologies (TSTT) Center, see `www.tstt-scidac.org`.

[6] B. Fryxell, K. Olson, P. Ricker, F. Timmes, M. Zingale, D. Lamb, P. MacNeice, R. Rosner, J. Truran, H. Tufo, FLASH: an adaptive-mesh hydrodynamics code for modeling astrophysical thermonuclear flashes, Astrophysical Journal Supplement 131 (2000) 273–334, see `www.asci.uchicago.edu`.

[7] R. H. et al., NWChem: A computational chemistry package for parallel computers, version 4.0.1, Tech. rep., Pacific Northwest National Laboratory (2001).

[8] C. Janssen, E. Seidl, M. Colvin, Object-oriented implementation of ab initio programs, ACS Symposium Series 592, Parallel Computers in Computational Chemistry .

[9] X. Z. Tang, G. Y. Fu, S. C. Jardin, L. L. Lowe, W. Park, H. R. Strauss, Resistive magnetohydrodynamics simulation of fusion plasmas, Tech. Rep. PPPL-3532, Princeton Plasma Physics Laboratory (2001).

[10] J. Shewchuk, Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator, in: Proceedings of the First Workshop on Applied Computational Geometry, ACM, Philadelphia, Pennsylvania, May 1996, pp. 124–133.

[11] B. M. Averick, R. G. Carter, J. J. Moré, The MINPACK-2 test problem collection, Tech. Rep. ANL/MCS-TM-150, Argonne National Laboratory (1991).

[12] A. M. Buraset, H. P. Langtangen, A comprehensive set of tools for solving partial differential equations: Diffpack, in: M. Daehlen, A. Tveito (Eds.), Numerical Methods and Software Tools in Industrial Methematics, Birkhauser Press, 1997, pp. 61–90.

[13] D. L. Brown, W. D. Henshaw, D. J. Quinlan, Overture: An object-oriented framework for solving partial differential equations on overlapping grids, in: Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, SIAM, 1999, pp. 58–67.

[14] V. Eijkhout, T. Chan, ParPre Web page, see `www.cs.utk.edu/~eijkhout/parpre.html`.

[15] R. Hornung, S. Kohn, The use of object-oriented design patterns in the SAMRAI structured AMR framework, in: Proceedings of the SIAM Workshop on Object-Oriented Methods for Inter-Operable Scientific and Engineering Computing, 1999.

[16] S. Balay, W. D. Gropp, L. C. McInnes, B. F. Smith, PETSc users manual, Tech. Rep. ANL-95/11 - Revision 2.1.1, Argonne National Laboratory (2001).

[17] S. Balay, W. D. Gropp, L. C. McInnes, B. F. Smith, Efficient management of parallelism in object oriented numerical software libraries, in: E. Arge, A. M. Bruaset, H. P. Langtangen (Eds.), Modern Software Tools in Scientific Computing, Birkhauser Press, 1997, pp. 163–202, see `www.mcs.anl.gov/petsc`.

[18] L. Freitag, M. Jones, C. Ollivier-Gooch, P. Plassmann, SUMAA3d Web page, see `www.mcs.anl.gov/sumaa3d`, Argonne National Laboratory.

[19] L. Freitag, M. Jones, P. Plassmann, Component integration for unstructured mesh algorithms and software, in: Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, SIAM, 1999, pp. 215–224.

[20] K. Buschelman, W. Gropp, L. McInnes, B. Smith, PETSc and Overture: Lessons learned developing an interface between components, in: Proceedings of the International Federation for Information Processing Working Conference on Software Architectures for Scientific Computing, Kluwer, Ottowa, Ontario, Canada, 2000, pp. 57–68, also available as Argonne preprint ANL/MCS-P858-1100.

[21] A. Hindmarsh et al., PVODE Web page, see `www.llnl.gov/CASC/PVODE`, Lawrence Livermore National Laboratory.

[22] S. Balay, W. D. Gropp, L. C. McInnes, B. F. Smith, A microkernel design for component-based parallel numerical software systems, in: Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, SIAM, 1999, pp. 60–69.

[23] J.-F. Remacle, O. Klaas, J. Flaherty, M. Shephard, Parallel algorithm oriented mesh database, in: Proceedings of the 10th International Meshing Roundtable, Sandia National Laboratories, October 2001, pp. 197–208.

[24] R. Clay, K. Mish, A. Williams, ISIS++ Web page, see `z.ca.sandia.gov/isis`.

[25] M. Heroux et al., Trilinos Web page, see `www.cs.sandia.gov/Trilinos`.

[26] R. Bramley, D. Gannon, T. Stuckey, J. Vilacis, E. Akman, J. Balasubramanian, F. Berg, S. Diwan, M. Govindaraju, The linear system analyzer, in *Enabling Technologies for Computational Science*, Kluwer, 2000.

[27] Robert Schneiders, Mesh generation and grid generation on the Web, see `www-users.informatik.rwth-aachen.de/~roberts/meshgeneration.html`.

[28] C. Szyperski, Component Software: Beyond Object-Oriented Programming, ACM Press, New York, 1998.

[29] R. Sessions, COM and DCOM: Microsoft's Vision for Distributed Objects, John Wiley & Sons, 1997, (see also `www.microsoft.com/com/about.asp`).

[30] R. Englander, Developing Java Beans, O'Reilly, 1997.

[31] R. Monson-Haefel, Enterprise JavaBeans, O'Reilly, 1999.

[32] R. Sistla, A. Dovi, P. Su, R. Shanmugasundaram, Aircraft design problem implementation under the Common Object Request Broker Architecture, in: 40th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference and Exhibit, 1999, pp. 1296–1305B.

[33] O. M. Group, CORBA Components, OMG TC Document orbos/99-02-05, 1999.

[34] J. Siegel, OMG overview: CORBA and the OMG in enterprise computing, Communications of the ACM 41 (10) (1998) 37–43.

[35] T. Epperly, S. Kohn, G. Kumfert, Component technology for high-performance scientific simulation software, in: Proceedings of the International Federation for Information Processing Working Conference on Software Architectures for Scientific Computing, Kluwer, Ottowa, Ontario, Canada, 2000, pp. 69–87.

[36] S. G. Parker, A component-based architecture for parallel multi-physics PDE simulations, proceedings of the 2002 International Conference on Computational Science, (to appear).

[37] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, M. Yechuri, A component based services architecture for building distributed applications, in: Proceedings of High Performance Distributed Computing, 2000.

[38] InDEPS Web page, `z.ca.sandia.gov/~indeps/`, Sandia Nat. Laboratories.

[39] K. Keahey, P. Beckman, J. Ahrens, Ligature: Component architecture for high-performance applications, International Journal of High-Performance Computing Applications 14 (4).

[40] K. Keahey, D. Gannon, PARDIS: A Parallel Approach to CORBA, in: Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computation, 1997, pp. 31–39.

[41] L. Freitag, W. Gropp, P. Hovland, L. McInnes, B. Smith, Infrastructure and interfaces for large-scale numerical software, in: H. R. Arabnia (Ed.), Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications, 1999, pp. 2657–2664.

[42] A. Geist, J. Kohl, P. Papadopoulos, CUMULVS: Providing fault tolerance, visualization and steering of parallel applications, The International Journal of High Performance Computing Applications 11 (3) (1997) 224–236.

[43] E. de Sturler, J. Hoeflinger, L. Kale, M. Bhandarkar, A new approach to software integration frameworks for multi-physics simulation codes, in: Proceedings of the International Federation for Information Processing Working Conference on Software Architectures for Scientific Computing, Kluwer, Ottowa, Ontario, Canada, 2000, pp. 87–105.

[44] C. René, T. Priol, G. Alléon, Code coupling using parallel CORBA objects, in: Proceedings of the International Federation for Information Processing Working Conference on Software Architectures for Scientific Computing, Kluwer, Ottowa, Ontario, Canada, 2000, pp. 105–118.

[45] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, J. Darlington, Optimisation of component-based applications within a grid environment, in: Proceedings of SC2001, 2001.

[46] H. Casanova, J. Dongarra, C. Johnson, M. Miller, Application specific tools, chapter 7, in *The Grid: Blueprint for a Future Computing Infrastructure*, Morgan Kaufmann Publishers, 1999.

[47] Common Component Architecture Draft Specification, see `www.cca-forum.org/specification/spec`.

[48] A. Cleary, S. Kohn, S. Smith, B. Smolinski, Language interoperability mechanisms for high-performance scientific computing, in: Proceedings of the SIAM Workshop on Object-Oriented Methods for Inter-Operable Scientific and Engineering Computing, 1999.

[49] S. Benson, L. C. McInnes, J. Moré, J. Sarich, TAO users manual, Tech. Rep. ANL/MCS-TM-242 - Revision 1.4, Argonne National Laboratory, see `www.mcs.anl.gov/tao` (2002).

[50] A. C. Hindmarsh, ODEPACK, a systematized collection of ODE solvers, in: Scientific Computing, 1993, pp. 55–64, vol. 1 of IMACS Transactions on Scientific Computation.

[51] S. Benson, L. C. McInnes, J. Moré, A case study in the performance and scalability of optimization algorithms, ACM Transactions on Mathematical Software 27 (2001) 361–376.

[52] J. Nocedal, S. J. Wright, Numerical Optimization, Springer-Verlag, New York, 1999.

[53] M. Sottile, C. Rasmussen, Automated component creation for legacy C++ and Fortran codes, in: Proceedings of the First International IFIP/ACM Working Conference on Component Deployment, Berlin, Germany, 2002, submitted.