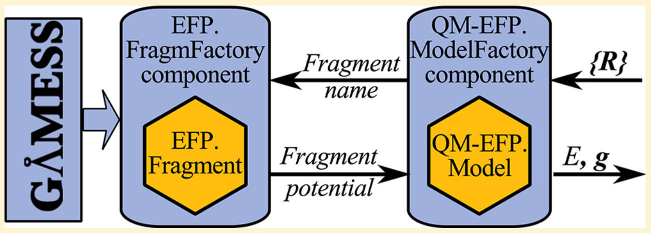
Design and Implementation of Scientific Software Components to Enable Multiscale Modeling: The Effective Fragment Potential (QM/EFP) Method

Alexander Gaenko, Theresa L. Windus, Masha Sosonkina, and Mark S. Gordon*

Ames Laboratory, Iowa State University, Ames, Iowa 50011, United States

ABSTRACT: The design and development of scientific software components to provide an interface to the effective fragment potential (EFP) methods are reported. Multiscale modeling of physical and chemical phenomena demands the merging of software packages developed by research groups in significantly different fields. Componentization offers an efficient way to realize new high performance scientific methods by combining the best models available in different software packages without a need for package readaptation after the initial componentization is complete. The EFP method is an efficient electronic structure theory based model potential that is suitable for predictive modeling of intermolecular interactions in large molecular systems, such as liquids, proteins, atmospheric aerosols, and nanoparticles, with an accuracy that is comparable to that of correlated *ab initio* methods. The developed components make the EFP functionality accessible for any scientific component-aware software package. The performance of the component is demonstrated on a protein interaction model, and its accuracy is compared with results obtained with coupled cluster methods.



INTRODUCTION

Scientific computing plays a major role in addressing global challenges. Scientific software needs to evolve constantly, adapting to new architectures (both homogeneous and hybrid), novel computer languages, state-of-the art algorithms, and various communication models. Specifically, multiscale modeling of physical and chemical phenomena often requires the merging of codes developed by specialist groups from significantly different disciplines and possibly written in different languages. Given these facts, component software development technology emerges as an attractive approach. Component technology allows the merging of multidisciplinary codes in a transparent and correct way, thereby ensuring maximum adaptability and maintainable correctness in the face of constant change. For example, complex quantum chemistry packages such as GAMESS,[1,2] NWChem,[3] and MPQC,[4] designed for high-performance scientific simulations, are component-aware. Exposing the quantum chemical methods contained in these packages as components enables interoperability and provides standard interfaces to the entire scientific community, thereby facilitating the development of multiscale modeling methods.

Scientific computing requires maximal performance of the available underlying hardware (including massively parallel platforms) and should support Fortran, complex arithmetic and multidimensional dynamically allocated arrays (preferably with arbitrary strides). It is important to provide a feasible migration strategy for existing software assets, incur minimal runtime overload, and be portable to most machines. Commercial component systems such as CORBA/CCM,[5−7] COM/COM+,[8] J2EE/EJB[9] and .Net[10] do not satisfy all of these criteria for scientific computations.[11,12]

The Common Component Architecture (CCA) paradigm was developed to create a standardized component framework that is targeted at high-performance scientific development.[13−15] CCA is not a software tool, but rather, it is a set of specifications for a framework that allow development, deployment, and interaction of independent scientific software components. At the base of the CCA is a language for specifying generic software interfaces called the Scientific Interface Definition Language (SIDL). A tool is required to read the SIDL files and generate the wrapper code that supports a uniform object oriented model across several languages in a single address space. In the CCA approach, the tool used has been Babel,[16] but in principle, other interface-definition language compilers[17] may be used instead, provided they have SIDL support. The component architecture specification written in SIDL dictates how CCA-compliant components interact with each other and with the underlying framework. A particular software tool that supports the deployment and execution of the CCA-compliant scientific software components represents an implementation of the CCA framework; the present work uses the Ccaffeine[18] implementation. Numerous software components with varying levels of maturity were developed to work with the CCA framework;[13] the main focus of the present paper is computational chemistry components.

Received: July 19, 2012
Published: October 19, 2012

The development of new scientific components is driven, first, by the desire to bring to the scientific community computational methods that were not universally available before, being, for example, restricted to a particular software package. The second reason is unification of access to the various implementations of a particular method. The latter reason is especially significant for computations comprising multiple time or length scales, because such computations would likely involve combining software packages developed in different research groups with different expertise. In addition, if multiple implementations are available for a component interface, then the best one can be chosen for a particular hardware architecture or operating system.

For example, understanding structure−activity relationships of large molecules (polymers, proteins, and enzymes), studying the aggregation of nanomaterials in solution, aerosol chemistry, mechanisms of chemical reactions, and interfacial (gas−liquid and liquid−solid) phenomena requires knowledge of intermolecular interactions. However, reliable quantum mechanical methods that take electron correlation effects into account (e.g., many body perturbation methods and coupled cluster methods) have prohibitively high CPU time scaling, $O(N^5)$−$O(N^7)$, where N is the total number of atomic basis functions. This scaling significantly limits the size of molecular systems that can be studied. An alternative feasible approach is to account for intermolecular interactions using accurate model potentials that have much lower scaling.

A promising approach is the hybrid quantum mechanical/effective fragment potential (QM/EFP) method.[19−28] The EFP method, which has been applied extensively, was originally formulated to describe chemical reactions in solution, in which the reacting species is described by QM and the solvent is described by EFPs. While the active region is treated accurately as a quantum system, its interaction with the effective fragments is described via one-electron terms in the Hamiltonian. One of the strengths of the EFP method is that the potential that describes the effective fragments is also obtained from a QM computation and thus can be systematically improved. The EFP method can also be used to study large clusters containing effective fragments of various kinds, as well as general intermolecular interaction phenomena. The latter do not require a QM component.

The long-term goal of the research described here is to employ component technology to enable a multiscale dynamics code to use an interaction potential evaluated using the electronic structure theory based EFP method. The method is provided by the freely available GAMESS[1,2] package and by the commercial QChem package.[29] The first task on the way to achieving this long-term goal, addressed in this paper, is to describe the design and implementation of the GAMESS-EFP components. Isolating the QM/EFP functionality as a set of components helps to maximize the adaptability of the EFP/QM code to changes in the high performance computing (HPC) environment, while maintaining the correctness of the implementation of new methods that depend on the code. In this paper, the implementation of both EFP and QM/EFP components is reported; the interface is available to any CCA-aware component package.

The paper is organized as follows: A short introduction to the Common Component Architecture is given in Section 2. An overview of a quantum chemistry component development is given in Section 3. A brief description of the QM/EFP method is given in Section 4. The design and implementation details of the EFP component models are given in Section 5. The results of the testing of the QM/EFP components are presented in Section 6.

2. THE COMMON COMPONENT ARCHITECTURE

A *component* is a self-contained piece of software that is capable of interacting with other components through a well-defined interface. The description of the components in the CCA is based on SIDL. A SIDL file contains definitions of classes and interfaces, grouped in packages. As a rule, a class implements one or more interfaces. Generally, any number of *objects* (data structures) that belong to the same class (the *instances* of the class) can be created; the process of creating an object is called the *class instantiation*. The declaration of a class and of all interfaces that a class implements defines a set of *object methods*: an object method is a function (or a subroutine) that can be applied to instances of a particular class to perform some operation on the instance, such as a query or a change of the object state.

To avoid confusion in the following text, it is necessary to strictly distinguish between object methods defined above (the functions or subroutines applicable to class instances) and quantum chemical methods (such as the coupled cluster or effective fragment potential methods).

After a SIDL file is translated into code in any of the supported languages (currently C, C++, Fortran, Java, or Python), class definitions become a callable skeleton code (the "server" code), which is extended by a component developer to do the actual computational work (usually, to call an existing legacy code or an application programming interface (API)). The interface definitions are translated into a "client" code; calling the client code transfers control to the "server" code of the class implementing the interface. It is important to note that the autogenerated "client" code and the corresponding developer-modified "server" code can be written in different languages. This way, the CCA object model can be mapped to any language, not necessarily an object-oriented one, and serves as a compatibility layer between languages. The CCA object model can be considered to be a language-independent generalization of the API approach.

After an instance of a class is created, a software developer utilizing the CCA object model (the "programmer") obtains not the object itself (the data structure in a computer memory), but a *reference* to it (conceptually similar to the address of the object in memory). Any number of references to the same object can be created, and the object methods are callable via the references to the object. When there are no more references in use, the object is automatically destroyed. Once the object is created and the reference to the object is obtained, there is no necessity for the programmer to know to what class the object belongs: it is sufficient to know the interface(s) that are implemented by the class. This way the interfaces (described in the SIDL files) are completely decoupled from the code implementing them (the callable code generated by the SIDL compiler from the class definitions), and the same interface can be implemented by different codes, thereby facilitating independent development and software interoperability.

A component in a CCA object model is an instance of the specialized class implementing the *Component* interface. The component class is selected by a programmer (e.g., using a graphical front-end) from a palette of available component classes, but the class instantiation is performed by the framework. Each component announces its interfaces (in this

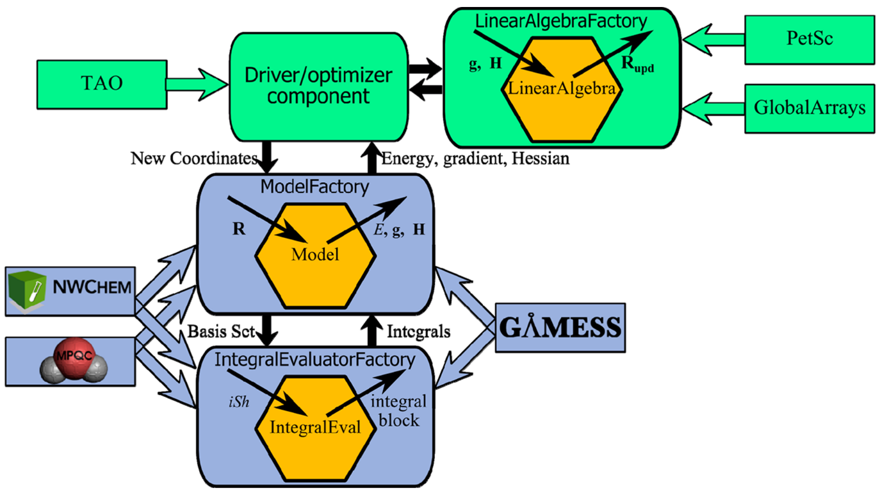


Figure 1. Schematic representation of quantum chemistry packages as components in the component architecture framework. The *ModelFactory* component, implemented by GAMESS, MPQC, or NWChem, calculates the energy (E), gradient ($\mathbf{g}$), Hessian ($\mathbf{H}$), given coordinates ($\mathbf{R}$) as input, and generates an object of the class *Model* encapsulating those values. The *ModelFactory* component uses the *IntegralEvaluatorFactory* component, implemented by GAMESS, NWChem, or MPQC. The *IntegralEvaluatorFactory* generates an object of the *IntegralEvaluator* class. The object returns an array of electron repulsion integrals, given shell numbers (*iSh*). The *LinearAlgebraFactory* component, implemented by the Global Arrays or PetSc, generates an object of the *LinearAlgebra* class, which is used by a driver component (implemented by TAO[34]) to compute new coordinates for the next geometry optimization step.

case called "provided" ports) to the framework during the initialization stage; the ports that are needed by the component from other components (the "used" ports) are also announced. When constructing a component system, the programmer connects the components through the framework so that each used port is connected to the corresponding provided port. A component obtains references to other components that are connected to it and calls object methods declared in their interfaces, thus accessing the functionality of the components via their provided ports.

An important kind of component is a factory. A factory component can create objects of certain classes and return references to them via the factory ports. The SIDL definition of a factory interface need not specify the class of the object created by the factory: only the interface implemented by the class is specified. There is also a "main" component (customarily called the "driver") from which the execution of the whole system begins. A typical usage pattern is as follows: the driver component obtains references to factory components to which it is connected; then, each factory component is called via its port. Each factory component creates a corresponding object, and a reference to the created object is then returned to the caller. In the process of creating the object, the factory component may call other components to which it is connected. Then, the driver operates on the created objects via the references that have been returned by the factories using the object methods declared in the corresponding interface definitions.

3. COMPONENT ARCHITECTURE AND CHEMISTRY

The collection of the chemistry-oriented CCA components, the CCA-Chem toolkit, defines generic interfaces and provides implementations of some components. The interfaces pertinent to quantum chemistry are grouped within the *Chemistry.QC* package to ensure a separate namespace (e.g., *Chemistry.QC.MoleculeInterface*). For brevity, in the remainder of this section, the prefix *Chemistry.QC* is omitted from the names of classes and interfaces. By convention, interfaces manipulating instances of a certain class (e.g., *Molecule*) have the suffix "*Interface*" (e.g., *MoleculeInterface*). Factory components producing objects of the class have the suffix "*Factory*" (e.g., *MoleculeFactory*) and can be manipulated through the corresponding interfaces (e.g., *MoleculeFactoryInterface*).

The *Model* class plays a central role in the CCA-Chem toolkit; the instances of the class represent the computational model of a molecular (or supramolecular) system in question. The *ModelInterface* implemented by the class provides the object methods *get_energy(), get_gradient(),* and *get_Hessian()* to request the energy, forces (gradients), and force constant matrix (Hessian), respectively. The instances of the *Model* class are generated by the *ModelFactory*, implementing the corresponding *ModelFactoryInterface*. The *ModelFactoryInterface* provides object methods to set the basis set, the level of theory, and the molecule (a collection of atom centers). While the concrete implementation of the *Model* class and the *ModelFactory* component should be provided by a quantum chemistry (QC) software package, the unified interfaces allow one to write a client code without any specific knowledge of the implementation of the quantum chemistry methods or the QC program input file format. The *ModelFactory* component can be considered to be a high-level CCA interface to the QC software package as a whole. MPQC, NWChem, and GAMESS developers have implemented the *ModelFactory* component based on the corresponding packages.[30−32] For example, in the GAMESS implementation, calling *ModelFactory* object methods is *equivalent* to making GAMESS parse its input file to set the level of theory, the basis set, the molecular geometry, and the type of calculation, in the sense that the internal state of the relevant parts of the GAMESS code becomes the same as if the data were read from an input file. Likewise, generating an instance of the *Model* class is *equivalent* to running the calculation, and requesting the properties of the *Model* object by calling the corresponding object methods is analogous to reading the GAMESS output file. (It should be noted that the above-mentioned equivalence does not mean that the *get_energy()* object method literally parses a GAMESS output file.)

In a similar way, instances of the *Molecule* class represent the collection of atom center coordinates of a molecule. The *MoleculeInterface* implemented by the class provides object methods to inquire about the types and positions of the atoms, and the *MoleculeFactory* component (implementing the corresponding *MoleculeFactoryInterface*) is used to produce instances of the *Molecule* class. A sample implementation of the *MoleculeInterface* (the *ChemistryCXX.Molecule* class) is provided by the CCA-Chem toolkit.[32]

To lay the foundation for the interactions between different QC software packages, in addition to the high-level interfaces described above, some lower-level interfaces are defined by the CCA-Chem toolkit. For example, the toolkit defines the *IntegralEvaluatorFactoryInterface* to be implemented by the factory components for 1- and 2-electron integral evaluators. Using these interfaces, it is possible to write a QC code that uses AO integrals from any QC package implementing the integral components. The *IntegralEvaluatorFactory* component was implemented by MPQC, NWChem, and GAMESS software developers based on the corresponding packages.[33]

The interaction between the CCA-Chem components within the CCA framework is schematically shown in Figure 1 for the example of a search for the molecular geometry with minimal energy (geometry optimization task). At the highest level, the task consists of finding a minimum value of an objective function (the energy) in the multidimensional space of its argument values (the molecular coordinates). The Toolkit for Advanced Optimization (TAO) software[34] is capable of performing this task. The optimization consists of a sequence of steps in the molecular coordinate space. In each step, the TAO component (implementing the Driver) requests the value of the objective function, as well as its gradient and Hessian. In the component scheme, other components are responsible for computing and providing those values. Here, new coordinates (**R**) are passed to a *ModelFactory* component, and the energy (E), gradient (**g**), and Hessian (**H**) evaluations are requested from the generated *Model* instance. By design, TAO puts a strong emphasis on the reuse of external software libraries (such as PETSc) for linear algebra operations. The Linear Algebra Factory component (implemented, e.g., by PETSc or Global Arrays) provides interfaces to its linear algebra classes, implementing highly efficient operations on large vectors and matrices. An instance of a linear algebra class generated by the corresponding Factory is used to compute the energy minimization step in coordinate space.

To generate the *Model* instance (encapsulating the energy, gradient and Hessian of the quantum system), the *ModelFactory* component performs, for example, a Hartree−Fock calculation. To obtain the two-electron integrals, it needs a connection to the *IntegralEvaluatorFactory*. A request to the factory provides the necessary initialization data (such as the basis set and positions of the centers) and returns an interface to an object of the *IntegralEvaluator* class that computes the required blocks of two-electron integrals.

An implementation of the linear algebra Factory by the Global Arrays or PETSc software packages, with an implementation of the QC model Factory and the integral evaluator Factory by the NWChem, MPQC or GAMESS packages, can be used for maximum performance for a given problem and a computational environment. The componentization of quantum chemistry software has been successfully used[35−40] to implement multilevel parallel calculations, to enable relativistic computations by providing access to an alternative integral library, to tie quantum chemical calculations to the computational quality of service framework, and to provide access to the NWChem implementation of the quantum mechanics/molecular mechanics (QM/MM) method.

4. EFFECTIVE FRAGMENT POTENTIAL METHOD

The EFP method is a first principles based model that incorporates the essential components of intermolecular interactions in the form of a model potential.[19−28] Only a brief overview, sufficient for understanding of the rest of the paper, is presented in this section.

The EFP parameters are generated from *ab initio* calculations of isolated molecules (the "fragments"); this approach places the EFP method into the family of sequential multiscale methods.[41] The system is regarded as consisting of one or more EFP fragments and, possibly, a quantum system; if the quantum system is present, the method is referred to as the QM/EFP method. Because the parameters of the effective fragment potential are determined by *ab initio* calculations, the EFP method is multiscale regardless of the presence or absence of a quantum system.

To understand the QM/EFP method, it is convenient to have in mind a typical application of the method; for example, a QM solute in an explicit EFP solvent. The QM/EFP Hamiltonian is written as follows:

$$\hat{H} = \hat{H}_{\mathrm{QM}} + V_{\mathrm{EFP}} \tag{1}$$

where $\hat{H}_{\mathrm{QM}}$ is the Hamiltonian of the isolated quantum system and V_{EFP} is the one-electron potential. The energy of the system is

$$E_{\mathrm{tot}} = \langle \Psi_{\mathrm{QM}} | \hat{H} | \Psi_{\mathrm{QM}} \rangle + E_{\mathrm{EFP}} \tag{2}$$

where Ψ_{QM} is the (perturbed) wave function of the quantum system and E_{EFP} is the energy due to fragment−fragment interactions.

The EFP method comes in two variants: EFP1 and EFP2. In the EFP1 variant, which is developed only for water molecules, the one-electron potential has the form:

$$V_{\mathrm{EFP1}} = V^{\mathrm{Coul}} + V^{\mathrm{Pol}} + V^{\mathrm{Rem}} \tag{3}$$

$$E_{\mathrm{EFP1}} = E^{\mathrm{Coul}} + E^{\mathrm{Pol}} + E^{\mathrm{Rem}} \tag{4}$$

where the superscript "Coul" corresponds to Coulombic (electrostatic) interactions, "Pol" corresponds to the polarization interactions, and "Rem" is the "remainder" term. The remainder term encompasses all interactions not taken into account by the Coulomb and polarization interactions. The remainder term is obtained by fitting the EFP1 potential to a Hartree−Fock, DFT or MP2 water-dimer potential after subtracting the Coulomb and polarization contributions. In the EFP2 variant the remainder potential is replaced by explicitly derived terms for the exchange repulsion ("XR"), charge transfer ("CT"), and dispersion ("Disp") contributions:

$$V_{\mathrm{EFP2}} = V^{\mathrm{Coul}} + V^{\mathrm{Pol}} + V^{\mathrm{XR}} + V^{\mathrm{CT}} + V^{\mathrm{Disp}} \tag{5}$$

$$E_{\mathrm{EFP2}} = E^{\mathrm{Coul}} + E^{\mathrm{Pol}} + E^{\mathrm{XR}} + E^{\mathrm{CT}} + E^{\mathrm{Disp}} \tag{6}$$

The exact analytic forms of the potentials can be found in the references cited in this section.

All of the components of the EFP potentials, except the polarization potential, are pairwise additive. Expressions for the polarization potential include dipole moments induced on the fragment by other fragments and the quantum system. The dipole moments, in turn, are determined by the fragment polarizability and the electric field due to the other fragments and the quantum system and thus have to be determined in a self-consistent manner.

Each contribution to the potential can be expanded as a sum of the potentials due to individual fragments:

$$V^{(x)} = \sum_A V_A^{(x)} \tag{7}$$

$$E^{(x)} = \sum_A \sum_B E_{AB}^{(x)} \tag{8}$$

where A and B enumerate fragments. Here and in the rest of this section, the superscript designating the nature of the potential is replaced by (x) for generality.

Computationally, each fragment is represented as a set of expansion points, and the potential induced by the fragment is the sum of the potentials induced by each of its points:

$$V_A^{(x)}(\mathbf{r}) = \sum_{i \in A} V^{\mathrm{pt},(x)}(\mathbf{r} - \mathbf{r}_i|\{a_{Ai}^{(x)}\}) \tag{9}$$

$$E_{AB}^{(x)} = \sum_{i \in A} \sum_{j \in B} E^{\mathrm{pt},(x)}(\mathbf{r}_i - \mathbf{r}_j|\{a_{Ai}^{(x)}\}, \{a_{Bj}^{(x)}\}) \tag{10}$$

where i, j enumerate fragment expansion points within fragments A, B, respectively; $V_A{}^{(x)}(\mathbf{r})$ is the potential of fragment A at location $\mathbf{r}$; $\mathbf{r}_i$ is the position of the i-th expansion point of the corresponding fragment; $V^{\mathrm{pt},(x)}$ and $E^{\mathrm{pt},(x)}$ are, respectively, the potential and the interfragment interaction energy associated with the fragment expansion points (as indicated by the "pt" superscript), and $\{a_{Ai}^{(x)}\}$ is the set of parameters for fragment A defining the potential induced by the i-th expansion point of the fragment.

Depending on the potential, the expansion points are positioned at atom positions, at bond midpoints, or at the fragment center of mass. As each fragment is considered to be rigid, the potential induced by a fragment is determined by the position of the center of mass and the orientation of the fragment. The values of the parameters are found by an *ab initio* quantum chemical calculation on the isolated molecule at the desired geometry.

5. DESIGN AND IMPLEMENTATION OF QM/EFP COMPONENTS

In the following discussion of the QM/EFP interface design and implementation, if not otherwise specified, all interfaces are defined within the *Chemistry.QC* package and all implementing classes are defined in the GAMESS package. To avoid confusion with similarly named classes and interfaces from the original CCA-Chem toolkit, in the following discussion the developed classes and interfaces are qualified with their outer package name, *Efp*. For example, the class referred as *Efp.ModelFactory* has the fully qualified name *GAMESS.Efp.ModelFactory*. Similarly, the fully qualified name of *Efp.ModelInterface* is *Chemistry.QC.Efp.ModelInterface*. By convention, package, class, and interface names start with capital letters, while object method names start with lower case letters.

A QM/EFP model is constructed by adding EFP fragments to a (possibly empty) quantum system. Thus, a QM/EFP model can be considered to be an extension of a usual quantum chemistry model, represented by *Chemistry.QC.Model*. Correspondingly, the new interfaces and component ports in the *Efp* package are designed as extensions (in the object-oriented sense) of the corresponding interfaces and ports of the CCA-Chem toolkit. That is, the new interfaces inherit object methods from their base interfaces and extend the base interfaces with relevant new object methods. The inheritance simplifies the SIDL descriptions of the interfaces and allows one to achieve a higher level of abstraction: e.g., a code unaware of new object methods available for an EFP model object still can treat the object as a QC model. Although the classes implementing the interfaces related by inheritance are not required to be related to each other, in the GAMESS-based implementation presented in this work the new classes and components are built as extensions of the classes and components developed earlier. The inheritance of the classes facilitates code reuse, improves development and runtime efficiency, and simplifies code management. Moreover, in the course of this work, several errors in the existing implementations of *Molecule* and *MoleculeFactory* classes in the CCA-Chem toolkit were found and corrected.

A new class *Efp.Model* representing the QM/EFP model (a quantum system surrounded by fragments) implements *Efp.ModelInterface*. The interface extends *ModelInterface* with object methods to request forces and torques acting on individual fragments, to facilitate the computation of EFP gradients. The corresponding factory component, *Efp.ModelFactory*, implements interface *Efp.ModelFactoryInterface*. The latter is an extension of *ModelFactoryInterface* with object methods to construct the QM/EFP model: *addFragment()* to add a fragment to the system and *removeFragment()* to remove a fragment. The position of the fragment is defined by a fictitious molecule specifying the coordinates of the expansion points of the fragment in the coordinate frame of the quantum system. The same chemical fragment can be added to the quantum system multiple times at different positions.

The fragment is represented as an instance of class *Efp.Fragment* implementing the corresponding *Efp.FragmentInterface*. From a programmer's point of view, a fragment is a reference to an immutable object that encapsulates all parameters necessary to fully define the EFP fragment: positions of multipole expansion points, polarizabilities, and so on ($\{a_{Ai}^{(x)}\}$ of eqs 9 and 10). Since a fragment cannot, by design, be changed once it is generated, the *Efp.FragmentInterface* does not provide any mutator object methods; the implementing class, however, defines several mutator object methods not intended to be called by a programmer. Those object methods are used internally by the C++ implementation of the fragment factory class (see below) to set up the properties of the object (the EFP fragment parameters) during the creation of the object. To facilitate retrieval of geometry information about the fragment, *Efp.FragmentInterface* also extends *MoleculeInterface*, but the implementation of *Efp.Fragment* does not allow calling of any of the object methods that change the molecule.

The fragments are generated by a factory component *Efp.FragmentFactory*, which provides an *Efp.FragmentFactoryInterface* port containing the object method *getFragment()*. The object method returns a reference to a newly created object of class *Efp.Fragment*, given a fragment name and a basis set name.

The scheme of interaction between the components and classes is shown in Figure 2: After retrieving a fragment from the *FragmentFactory* component via the *getFragment()* object method, the *Driver* adds the fragment (at different positions) to the quantum system by calling the *addFragment()* object method on the reference to the QM-EFP *ModelFactory* component; then, the *Driver* requests the generated *Model* object from the *ModelFactory* by calling the *get_model()* object method. The *Model* object then evaluates the energy (or any other property) of the QM-EFP system at the request of the *Driver*.

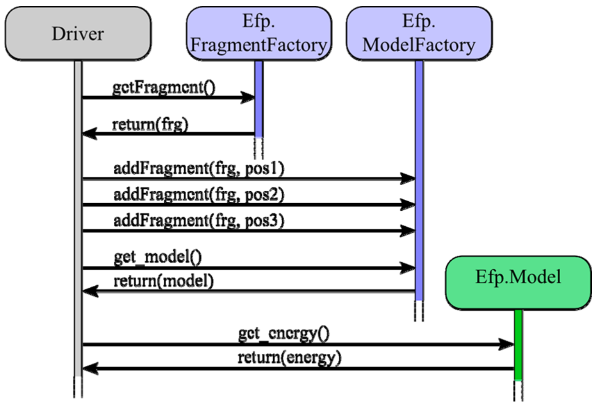


Figure 2. Sequence diagram illustrating the usage of EFP components. The *Driver* component requests a fragment from the *FragmentFactory* component. Then, the *Driver* adds the fragment (at different positions) to the quantum system by calling the QM-EFP *ModelFactory* component and requests the generated *Model* object from the *ModelFactory*. The *Model* object then evaluates the energy of the QM-EFP system at the request of the *Driver*.

The fragment factory maintains a simple persistent database, into which the fragments are put by *storeFragment()*. A fragment is generated in a separate run by an *ab initio* computation: The interface *Efp.ModelFactoryInterface* provides the *getFragment()* object method returning a reference to a fragment generated from the given molecular geometry and basis set setup in the factory object (see Figure 3): After setting the molecular geometry, the *Driver* component requests an *Efp.Fragment* object from the *Efp.ModelFactory* component via its *getFragment()* object method (note that this object method is distinct from the object method of the same name that is provided by the *Efp.FragmentFactory* component); in the process of generating the fragment, the *Efp.ModelFactory* component uses the *MolecularBasisFactory* component to request a molecular basis set. Likewise, the *MolecularBasisFactory* component uses the *AtomicBasisFactory* component to request basis sets of individual atoms. The EFP fragment, generated by the *ModelFactory* component, is returned to the *Driver*; then the *Driver* can store the generated fragment in the *FragmentFactory* component by calling its *storeFragment()* object method for future use.

The interface design choice to generate a fragment by request to the model factory component, rather than via the object of the *Efp.Model* class, may seem to be less intuitive at first sight. The rationale for the design choice is that a fragment can be considered as a (coarse-grained) representation of the quantum system (that is, a kind of model, to be generated by the model factory), rather than as a property of the quantum system (like energy or gradient).

Alternatively, the factory allows one to create fragments "from scratch" by specifying individual fragment parameters $\{a_{Ai}^{(x)}\}$ (positions and values of multipoles, polarizabilities, etc.) explicitly and passing them to corresponding object methods in *Efp.FragmentFactoryInterface*.

For use by the model factory, a molecular basis set class and molecular basis set factory component are implemented in C++ as a part of the CCA-Chem toolkit; the implementation is based on the interface *GaussianBasis.MolecularInterface*. Although the latter was defined in the CCA-Chem toolkit, no stand-alone implementations of the class and the factory were available. The molecular basis set factory component uses a newly developed atomic basis set factory component to generate objects of the atomic basis set class. The corresponding interfaces *AtomicInterface* and *AtomicFactoryInterface* are added to the *Chemistry.QC.GaussianBasis* package of the CCA-Chem toolkit. The developed atomic basis set component has the capability of reading atomic basis set files of various formats; new formats can be easily added by extending the base "reader" class in the implementation of the component. The molecular basis set component allows any

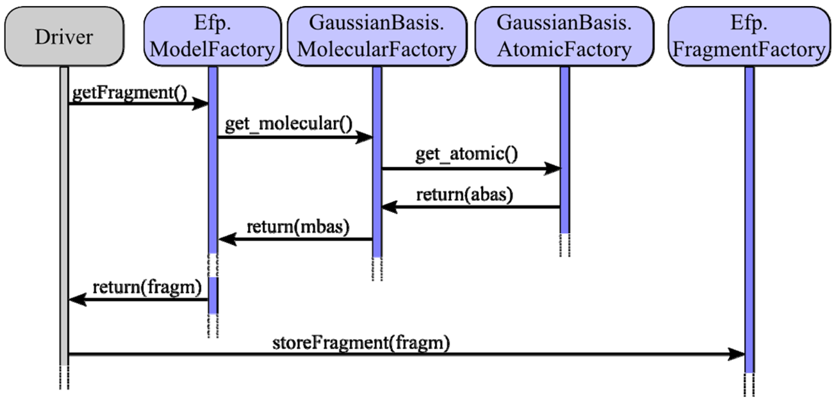


Figure 3. Sequence diagram illustrating fragment generation from a quantum system. The *Driver* component requests a *Fragment* object from the *ModelFactory* component, which in turn uses the *MolecularBasisFactory* to request a molecular basis. The *MolecularBasisFactory*, likewise, uses the *AtomicBasisFactory* to request an atomic basis. Then the *ModelFactory* generates an EFP fragment and returns it to the *Driver*; the *Driver* stores the generated fragment in the *FragmentFactory* component.

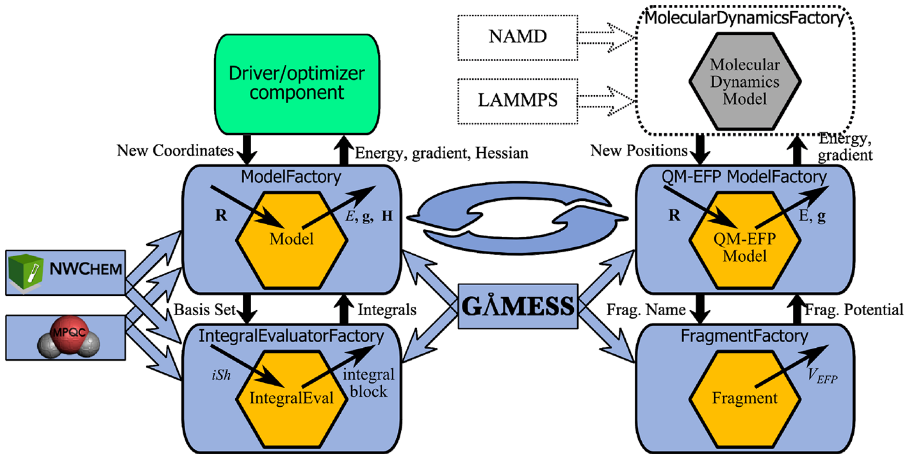


Figure 4. Usage of QC components and EFP components (cf. Figure 1, with nonessential elements omitted). The QM-EFP ModelFactory and the QM-EFP Model are fully interchangeable with the old QC ModelFactory and the QC Model: given the positions **R** of atoms and fragments, the new ModelFactory returns energy E and gradient **g** encapsulated in the Model. The QM-EFP ModelFactory uses fragments retrieved by name from the FragmentFactory, thus obtaining the fragment potential V_{EFP}. A hypothetical *MolecularDynamicsFactory* component (implemented on the basis of a Molecular Dynamics program) can use the new QM-EFP Model, thus adding the EFP potential to the Molecular Dynamics.

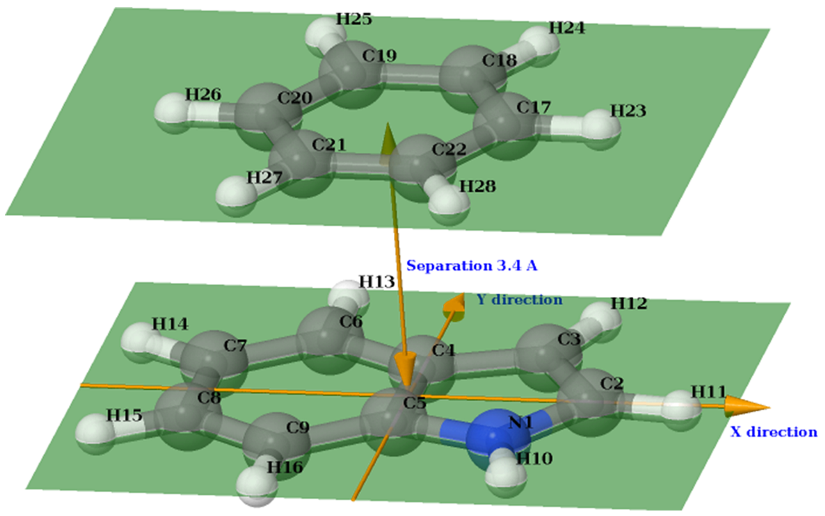


Figure 5. Coordinate system used to construct the indole−benzene sandwich complex potential energy surface and the molecular system at zero displacement. The molecules lie in parallel planes at 3.4 Å separation. The center of origin is at the midpoint of the C4−C5 bond of indole; the *Y*-axis positive direction is along the C4−C5 bond toward the C4 atom; the *X*-axis positive direction is perpendicular to the *Y*-axis toward the C2 atom. The displacement is measured between the center of origin and the center of mass of the benzene molecule.

model factory component implemented by a QC software package to read basis sets in any format supported by the basis set component.

A difficulty in implementing GAMESS CCA components is that the mainstream GAMESS code is designed to run as a standalone executable, not as a callable library, as required by the CCA framework. To work around this difficulty, the previous GAMESS-based implementation of the CCA components and classes (*ModelFactory* and *Model*) used CCA-callable "wrapper" Fortran subroutines that essentially duplicated the existing GAMESS code with minimal modifications. It turned out to be very inconvenient and error-prone to keep the "wrapper" code in sync with the evolving mainstream GAMESS code.

To greatly simplify the maintenance of the wrapper code, a template-driven autogeneration approach is employed. Relevant sections of the mainstream GAMESS code were marked with special comments, and a Perl script was used to generate the wrapper functions from templates by inserting the marked sections of the mainstream code into the designated places in the templates.

The usage of the new EFP components within the CCA framework is shown in Figure 4; it is analogous to the usage of the CCA-Chem components described in Section 3 (Figure 1). The QM-EFP *ModelFactory* receives the positions of the atoms in the QM system and the positions and orientations of the fragments; the fragments are obtained from the *FragmentFactory*. When generating the object of the *Model* class, the QM-EFP *ModelFactory* takes into account the fragment potential V_{EFP} that it obtains from the parameters of the fragments. It is important to note that, as the *EFP.ModelFactoryInterface* is derived from the *ModelFactoryInterface*, a QM/EFP model can transparently replace a QM model: given the positions **R** of atoms and fragments, the new Model returns energy *E* and gradient **g** for the QM system (in the presence of EFP fragments). In future research, a component for performing molecular dynamics simulations with the EFP components by calling the code provided by the NAMD[42] or LAMMPS[43] software packages will be designed and implemented.

6. APPLICATION OF QM/EFP COMPONENT: INDOLE–BENZENE INTERACTIONS

To test the newly developed EFP2 components, a potential energy surface for the indole-benzene sandwich $\pi-\pi$ type interactions was computed. The $\pi-\pi$ interactions are the most prevalent noncovalent interactions between aromatic subunits of nucleic acids and proteins, and the noncovalent interactions are a major factor determining the structures of those biomolecules. The indole and benzene molecules are commonly used to model aromatic side chains of the tryptophan and phenylalanine amino acids, respectively.

The sandwich indole–benzene structure recently studied by Geng et al.[44] was taken as a model chemical system. The EFP fragments for both indole and benzene were generated with the 6-311++G(3d,p) basis set, with the fragment geometries being set to those reported in the work of Geng et al.[44] for indole and benzene.

The potential energy surface for the indole–benzene interactions in a sandwich conformation was constructed using the EFP components. The initial conformation and the coordinate system used are shown in Figure 5. The distance between the planes of the molecules was kept at 3.4 Å, as in ref 44, while displacing the benzene in the *X* and *Y* directions relative to the initial conformation. A grid of 10000 (100 × 100) displacement points was used, and for each grid point, the EFP method was called via the interfaces described above. Thanks to the interlanguage component model of the CCA, the driver component constructing the rectangular grid in the coordinate space can be kept simple, comprising only a few dozen lines of C++ code (most of them validating the user input).

The computed potential energy surface, projected onto the indole molecule skeleton and color-coded according to the interaction energy, is shown in Figure 6. From the figure, it can be immediately seen that the potential energy surface exhibits a strong maximum near the center of the benzene ring, a less pronounced maximum at the indole ring, and a deep minimum near the indole nitrogen atom outside the ring; similar results were obtained by Geng et al.[44] using the SCS-MP2 method.[45] The position of the energy minimum and the interaction energy at the minimum obtained in the present work are shown in Table 1 along with the results reported in ref 44. The minimum energy structure (the view along the molecule plane normal) is shown in Figure 7. Although it is possible to systematically improve the minimum geometry estimate (e.g., by increasing the grid resolution or expanding the basis set used in generating the fragments), it is not deemed necessary here. The goal of the present research is to develop a scientific software component to access the EFP method rather than validating the method itself; the latter task has been addressed in numerous other works—see recent works of Smith et al.[46,47] and references therein.

7. CONCLUSIONS

In this paper, the design and implementation of components to access molecular basis sets, EFP1, EFP2, QM/EFP(1), and QM/EFP(2) methods are reported. Classes and components for generating and representing the QM/EFP model

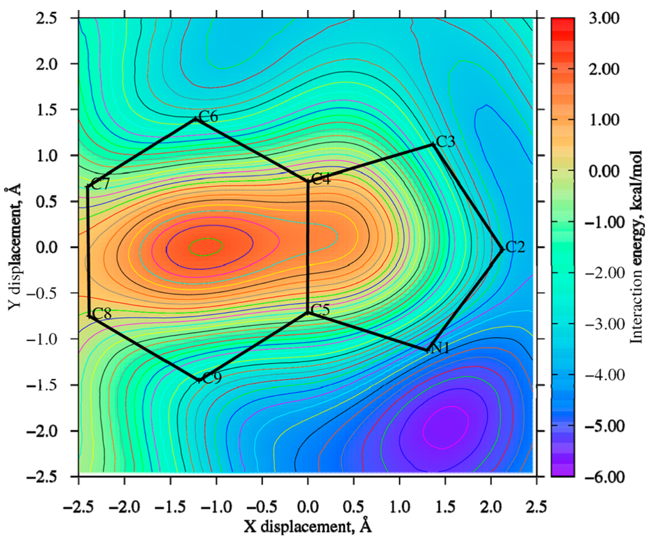


Figure 6. Potential energy surface of the indole–benzene sandwich structure (the interaction energy as a function of the two-dimensional displacement of the benzene molecule relative to the indole molecule), constructed using the EFP components. The surface is projected onto the indole skeleton and color-coded according to the interaction energy. The origin (zero displacement) corresponds to the middle of the C4–C5 bond of indole. A deep energy minimum can be identified in the vicinity of the N1 nitrogen atom.

Table 1. Geometry and Interaction Energy of the Indole–Benzene Sandwich Structure at the Minimum Energy Conformation (Parallel Separation is 3.4 Å)[a]

	EFP2[b]	SCS-MP2[c]	CCSD(T)//SCS-MP2[c]
X displacement, Å	1.5	1.3	1.3
Y displacement, Å	−2.0	−1.8	−1.8
interaction energy, kcal/mol	−5.7	−4.2	−4.6

[a]The SCS-MP2 and CCSD(T) energies and geometries are given as reported in ref 44. [b]6-31G(3d,p) basis. [c]cc-pVDZ basis; reported in ref 44

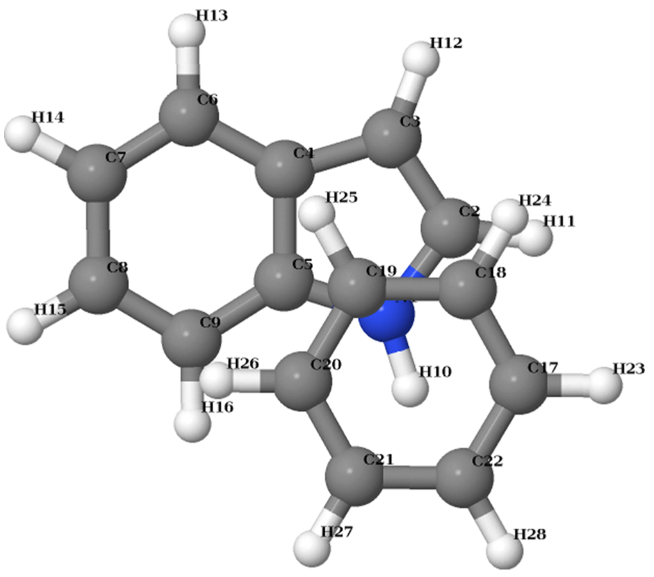


Figure 7. Indole–benzene sandwich structure corresponding to a local minimum of the interaction energy. Parallel separation is 3.4 Å.

(*Efp.ModelFactory*, *Efp.Model*), as well as classes and components for generating and representing the EFP fragments (*Efp.FragmentFactory*, *Efp.Fragment*) have been added to the set of available components along with the SIDL declarations of the corresponding interfaces. The EFP2 potential is applicable to any molecular environment and improvable in a systematic manner. The GAMESS component was modified and the code layout was redesigned to simplify merging with future GAMESS versions. A potential energy surface using 10000 points was generated with the EFP2 method, and the PES features and the location of the energy minimum are found to be in a good agreement with the CCSD(T)//SCS-MP2 results. The developed components are available for any component-aware packages using Babel. Current efforts are to design and implement components to interface an efficient molecular dynamics or Monte Carlo code within the component framework. The applicability of the QM/EFP method has recently been extended to study excited state chemistry.[29,48,49] Together with the proposed MC and MD components, the developed QM/EFP components will enable realistic and predictive modeling of excited state dynamics photochemical reactions[50,51] occurring in biological systems and in the atmosphere including effects of large molecular environment.

■ AUTHOR INFORMATION

Corresponding Author

*E-mail: mark@si.msg.chem.iastate.edu.

Notes

The authors declare no competing financial interest.

Software availability. The Common Component Architecture packages[13−15] are available at http://sourceforge.net/projects/cca-forum/; the CCA-Chem components[32] are available via anonymous CVS at cca-forum.org/cvs/cca-chem (repository cca-chem-generic2); the GAMESS-specific components, including the EFP extensions described here, are available from the authors by request.

■ ACKNOWLEDGMENTS

This work was supported by a U.S. Department of Energy Scientific Discovery Through Advanced Computing (SciDAC) grant to the Ames Laboratory. Helpful discussions with Dr. David Bernhold are gratefully acknowledged.

■ REFERENCES

(1) Schmidt, M. W.; Baldridge, K. K.; Boatz, J. A.; Elbert, S. T.; Gordon, M. S.; Jensen, J. H.; Koseki, S.; Matsunaga, N.; Nguyen, K. A.; Su, S.; Windus, T. L.; Dupuis, M.; Montgomery, J. A., Jr. *J. Comput. Chem.* **1993**, *14*, 1347−1363.

(2) Gordon, M.; Schmidt, M. Advances in electronic structure theory: GAMESS a decade later. In *Theory and Applications of Computational Chemistry*; Dykstra, C. E., Frenking, G., Kim, K. S., Scuseria, G. E., Eds.; Elsevier: Amsterdam, 2005; Chapter 41.

(3) Bylaska, E. J.; de Jong, W. A.; Govind, N.; Kowalski, K.; Straatsma, T. P.; Valiev, M.; Wang, D.; Apra, E.; Windus, T. L.; Hammond, J.; Nichols, P.; Hirata, S.; Hackler, M. T.; Zhao, Y.; Fan, P.-D.; Harrison, R. J.; Dupuis, M.; Smith, D. M. A.; Nieplocha, J.; Tipparaju, V.; Krishnan, M.; Wu, Q.; van Voorhis, T.; Auer, A. A.; Nooijen, M.; Brown, E.; Cisneros, G.; Fann, G. I.; Fruchtl, H.; Garza, J.; Hirao, K.; Kendall, R.; Nichols, J. A.; Tsemekhman, K.; Wolinski, K.; Anchell, J.; Bernholdt, D.; Borowski, P.; Clark, T.; Clerc, D.; Dachsel, H.; Deegan, M.; Dyall, K.; Elwood, D.; Glendening, E.; Gutowski, M.; Hess, A.; Jaffe, J.; Johnson, B.; Ju, J.; Kobayashi, R.; Kutteh, R.; Lin, Z.; Littlefield, R.; Long, X.; Meng, B.; Nakajima, T.; Niu, S.; Pollack, L.; Rosing, M.; Sandrone, G.; Stave, M.; Taylor, H.; Thomas, G.; van Lenthe, J.; Wong, A.; Zhang, Z. *NWChem, A Computational Chemistry Package for Parallel Computers*, Version 5.1; Pacific Northwest National Laboratory: Richland, WA, 2007.

(4) Janssen, C. L.; Nielsen, I. B.; Leininger, M. L.; Valeev, E. F.; Kenny, J. P.; Seidl, E. T. *The Massively Parallel Quantum Chemistry Program (MPQC)*, Version 3; Sandia National Laboratories, Livermore, CA, 2008.

(5) *CORBA Specification, Version 3.2: CORBA Interfaces*; Object Management Group: Needham, MA, 2011, formal/2011-11-01. Available online: http://www.omg.org/spec/CORBA/3.2/Interfaces/PDF/ (accessed 10/25/2012).

(6) *CORBA specification, Version 3.2: CORBA Interoperability*; Object Management Group: Needham, MA, 2011, formal/2011-11-02. Available online: http://www.omg.org/spec/CORBA/3.2/Interoperability/PDF (accessed 10/25/2012).

(7) *CORBA specification, Version 3.2: CORBA Component Model*; Object Management Group: Needham, MA, 2011, formal/2011-11-03. Available online: http://www.omg.org/spec/CORBA/3.2/Components/PDF (accessed 10/25/2012).

(8) *COM+ Developer's Reference Library*; Iseminger, D., Ed.; Windows Programming Reference Series; Microsoft Press: Redmond, WA, 2000.

(9) *Enterprise JavaBeans 3.0*. http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html (accessed 10/25/2012).

(10) Microsoft.net homepage. http://www.microsoft.com/net (accessed 10/25/2012).

(11) Armstrong, R.; Kumfert, G.; McInnes, L. C.; Parker, S.; Allan, B.; Sottile, M.; Epperly, T.; Dahlgren, T. *Int. J. Concurrency Comput.: Pract. Exper.* **2006**, *18*, 215−229.

(12) Cleary, A.; Kohn, S.; Smith, S. G.; Smolinski, B. Language interoperability mechanisms for high-performance scientific applications. *SIAM Workshop on Object-Oriented Methods for Inter-operable Scientific and Engineering Computing*, Yorktown Heights, NY, 1998, LLNL document UCRL-JC-131823.

(13) Common Component Architecture Forum. http://www.cca-forum.org/ (accessed 10/25/2012).

(14) Bernholdt, D. E.; Allan, B. A.; Armstrong, R.; Bertrand, F.; Chiu, K.; Dahlgren, T. L.; Damevski, K.; Elwasif, W. R.; Epperly, T. G.; Govindaraju, M.; Katz, D. S.; Kohl, J. A.; Krishnan, M.; Kumfert, G.; Larson, J. W.; Lefantzi, S.; Lewis, M. J.; Malony, A. D.; Mclnnes, L. C.; Nieplocha, J.; Norris, B.; Parker, S. G.; Ray, J.; Shende, S.; Windus, T. L.; Zhou, S. *Int. J. High Perform. Comput. Appl.* **2006**, *20*, 163−202.

(15) McInnes, L. C.; Allan, B. A.; Armstrong, R.; Benson, S. J.; Bernholdt, D. E.; Dahlgren, T. L.; Diachin, L. F.; Krishnan, M.; Kohl, J. A.; Larson, J. W.; Lefantzi, S.; Nieplocha, J.; Norris, B.; Parker, S. G.; Ray, J.; Zhou, S. Parallel PDE-based simulations using the common component architecture. In *Numerical Solution of PDEs on Parallel Computers*; Bruaset, M., Bjørstad, P., Tveito, A., Eds.; Springer-Verlag: Berlin/Heidelberg, 2006; Vol. *51*, pp 327−384. Also available as Argonne National Laboratory technical report ANL/MCS-P1179-0704.

(16) Dahlgren, T.; Epperly, T.; Kumfert, G.; Leek, J. *Babel User's Guide*, Version Babel-0.99.0 ed.; CASC, Lawrence Livermore National Laboratory: Livermore, CA, 2006.

(17) *Open Management Group IDL: Details, Web document*. http://www.omg.org/gettingstarted/omg_idl.htm (accessed 10/25/2012).

(18) Allan, B.; Armstrong, R.; Lefantzi, S.; Ray, J.; Walsh, E.; Wolfe, P. *Ccaffeine—A CCA Component Framework for Parallel Computing*; CCA: , 2003. Available online: http://www.cca-forum.org/ccafeine (accessed 10/25/2012).

(19) Day, P. N.; Jensen, J. H.; Gordon, M. S.; Webb, S. P.; Stevens, W. J.; Krauss, M.; Garmer, D.; Basch, H.; Cohen, D. *J. Chem. Phys.* **1996**, *105*, 1968−1986.

(20) Jensen, J. H.; Gordon, M. S. *J. Chem. Phys.* **1998**, *108*, 4772−4782.

(21) Jensen, J. H. *J. Chem. Phys.* **2001**, *114*, 8775−8783.

(22) Jensen, J. H. *J. Chem. Phys.* **1996**, *104*, 7795−7796.

(23) Gordon, M. S.; Freitag, M. A.; Bandyopadhyay, P.; Jensen, J. H.; Kairys, V.; Stevens, W. J. *J. Phys. Chem. A* **2001**, *105*, 293−307.

(24) Adamovic, I.; Gordon, M. S. *Mol. Phys.* **2005**, *103*, 379−387.

(25) Li, H.; Gordon, M. S. *J. Chem. Phys.* **2006**, *124*, 214108–214123.
(26) Adamovic, I.; Freitag, M. A.; Gordon, M. S. *J. Chem. Phys.* **2003**, *118*, 6725–6732.
(27) Slipchenko, L. V.; Gordon, M. S. *J. Comput. Chem.* **2007**, *28*, 276–291.
(28) Gordon, M.; Slipchenko, L.; Li, H.; Jensen, J. *Ann. Rep. Comp. Chem.* **2007**, *3*, 177–193.
(29) Slipchenko, L. V. *J. Phys. Chem. A* **2010**, *114*, 8824–8830.
(30) Krishnan, M.; Alexeev, Y.; Windus, T.; Nieplocha, J. Multilevel parallelism in computational chemistry using common component architecture and global arrays. *Proceedings of Supercomputing*, Seattle, WA, 2005.
(31) Peng, F.; Wu, M.-S.; Sosonkina, M.; Kendall, R. A.; Schmidt, M. W.; Gordon, M. S. Coupling GAMESS via standard interfaces. *Proceedings of HPCGECO/Compframe 2006 Joint Workshop*, Paris, France, 2006.
(32) Kenny, J. P.; Benson, S. J.; Alexeev, Y.; Sarich, J.; Janssen, C. L.; Mcinnes, L. C.; Krishnan, M.; Nieplocha, J.; Jurrus, E.; Fahlstrom, C.; Windus, T. L. *J. Comput. Chem.* **2004**, *25*, 1717–1725.
(33) Kenny, J. P.; Janssen, C. L.; Valeev, E. F.; Windus, T. L. *J. Comput. Chem.* **2008**, *29*, 562–577.
(34) Munson, T.; Sarich, J.; Wild, S.; Benson, S.; McInnes, L. C. *TAO 2.0 Users Manual*; Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 2012; ANL/MCS-TM-322. Available online: http://www.mcs.anl.gov/tao.
(35) Alexeev, Y.; Allan, B. A.; Armstrong, R. C.; Bernholdt, D. E.; Dahlgren, T. L.; Gannon, D.; Janssen, C. L.; Kenny, J. P.; Krishnan, M.; Kohl, J. A.; Kumfert, G.; McInnes, L. C.; Nieplocha, J.; Parker, S. G.; Rasmussen, C.; Windus, T. L. *J. Phys.: Conf. Ser.* **2005**, *16*, 536–540.
(36) Janssen, C. L.; Kenny, J. P.; Nielsen, I. M. B.; Krishnan, M.; Gurumoorthi, V.; Valeev, E. F.; Windus, T. L. *J. Phys.: Conf. Ser.* **2006**, *46*, 220–228.
(37) Kenny, J. P.; Janssen, C. L.; Gordon, M. S.; Sosonkina, M.; Windus, T. L. *Sci. Program.* **2008**, *16*, 287–296.
(38) Wu, M.-S.; Mori, H.; Bentz, J.; Windus, T. L.; Netzloff, H.; Sosonkina, M.; Gordon, M. S. Constructing a performance database for large-scale quantum chemistry packages. *Proceedings of High Performance Computing and Simulation Symposium*, Ottawa, Canada, 2008; pp 393–400.
(39) Li, L.; Kenny, J.; Wu, M.-S.; Huck, K.; Gaenko, A.; Gordon, M. S.; Janssen, C. L.; McInnes, L. C.; Netzloff, H. M.; Norris, B.; Windus, T. L. Adaptive application composition in quantum chemistry. *Quality of Software Architectures (QoSA '09) Int. Conf., E.*; Stroudsburg University, PA, 2009.
(40) Gulabani, T.; Sosonkina, M.; Gordon, M.; Janssen, C.; Kenny, J.; Netzloff, H.; Windus, T. Development of high performance scientific components for interoperability of computing packages. *Proceedings of the 2009 Spring Simulation Multiconference*, San Diego, CA, 2009.
(41) Makov, G.; Gattinoni, C.; De Vita, A. *Modell. Simul. Mater. Sci. Eng.* **2009**, *17*, 084008.
(42) NAMD Web-site. http://www.ks.uiuc.edu/Research/namd/ (accessed 10/25/2012).
(43) LAMMPS Web-site. http://lammps.sandia.gov (accessed 10/25/2012).
(44) Geng, Y.; Takatani, T.; Hohenstein, E. G.; Sherrill, C. D. *J. Phys. Chem. A* **2010**, *114*, 3576–3582.
(45) Grimme, S. *J. Chem. Phys.* **2003**, *118*, 9095–9102.
(46) Smith, Q. A.; Gordon, M. S.; Slipchenko, L. V. *J. Phys. Chem. A* **2011**, *115*, 4598–4609.
(47) Smith, Q. A.; Gordon, M. S.; Slipchenko, L. V. *J. Phys. Chem. A* **2011**, *115*, 11269–11276.
(48) Sok, S.; Willow, S. Y.; Zahariev, F.; Gordon, M. *J. Phys. Chem. A* **2011**, *115*, 9801–9809.
(49) Ghosh, D.; Isayev, O.; Slipchenko, L. V.; Krylov, A. I. *J. Phys. Chem. A* **2011**, *115*, 6028–6038.
(50) Devarajan, A.; Gaenko, A.; Lindh, R.; Malmqvist, P.-Å. *Int. J. Quantum Chem.* **2009**, *109*, 1962–1974.
(51) Gaenko, A. V.; Devarajan, A.; Gagliardi, L.; Lindh, R. *Theor. Chem. Acc.* **2006**, *118*, 271–279.