SAND2007-0258J

Components for Integral Evaluation in Quantum Chemistry

Joseph P. Kenny,[1] Curtis L. Janssen,[1] Edward F. Valeev,[2] and Theresa L. Windus[3]

[1]*Scalable Computing Research and Design, Sandia National Laboratories,*

MS 9158, P.O. Box 969, Livermore, CA 94551-0969

[2]*Department of Chemistry, Virginia Tech, 403 Davidson Hall,*

Blacksburg, VA 24061-0001

[3]*Department of Chemistry, Iowa State University, 1605 Gilman Hall,*

Ames, IA 50011

Abstract

Sharing low-level functionality between software packages enables more rapid development of new capabilities and reduces the duplication of work amongst development groups. Using the component approach advocated by the Common Component Architecture Forum, we have designed a flexible interface for sharing integrals between quantum chemistry codes. Implementation of these interfaces has been undertaken within the Massively Parallel Quantum Chemistry package, exposing both the IntV3 and Cints integrals packages to component applications. Benchmark timings for Hartree-Fock calculations demonstrate that the overhead due to the added interface code varies significantly, from less than 1% for small molecules with large basis sets, to nearly 10% for larger molecules with smaller basis sets. Correlated calculations and density functional approaches encounter less severe performance overheads of less than 5%. While these overheads are acceptable, additional performance losses occur when arbitrary implementation details, such as integral ordering within buffers, must be handled. Integral reordering is observed to add an additional overhead as large as 12%, making the agreement on and adoption of standards a requirement for improved future performance of interoperable codes.

Key words: electronic structure, integral, component, software development

1 Introduction

Historically, efforts to combine functionality from quantum chemistry software packages have been limited in scope and consisted of package-specific, one-to-one solutions. Such one-to-one code integration suffers from poor scaling of programming effort; the effort to share features from n packages grows as $O(n^2)$. Consequently, the integration of codes from various packages is frequently considered tedious and not worthwhile, regardless of any advances in methods and capabilities which might become available. Considering the large number of noncommercial packages existing within the quantum chemistry community, each with unique capabilities and deficiencies, this situation is unfortunate.

The future high performance computers on which quantum chemistry packages will run may depend on advanced processor architectures. Regardless of whether such systems contain co-processors which augment conventional general purpose processors or contain entirely unfamiliar processor architectures,[1,2] wide adoption of such technology will be impractical without a community code base. It is clear that a scalable approach to creating interoperable software must be adopted both to enable rapid development of advanced quantum chemical methods and to ensure such methods run reliably on the latest hardware.

Component-based software approaches break up complex tasks into loosely coupled subproblems, encouraging the definition of standardized interfaces and enabling collaboration between research groups. The Common Component Architecture (CCA) Forum is a consortium dedicated to the development and adoption of a component architecture for scientific software.[3–5] Work within the CCA includes the development of standards and middleware,[6–12] as well as component toolkits within numerous scientific domains. Development groups adopting component technology for scientific codes span such diverse domains as optimization and linear algebra, combustion and quantum chemistry, accelerator design, fire and explosives modeling, and climate simulation.[13–17]

Previously, several of the authors participated in work towards a quantum chemistry component toolkit, using development practices advocated by the CCA Forum.[16–18] Using CCA components to manage multiple levels of parallelism, significant improvements in machine utilization were demonstrated.[18] Through the development of an application for molecular structure optimization, it was demonstrated that domain scientists adopting CCA approaches are able to integrate software packages from multiple scientific disciplines.[16] Molecular structure optimization required the functionality of each major mathematics and chemistry package to be encapsulated in a set of components and classes. There was, however, no attempt to implement low-level interfaces between chemistry packages. Thus, while *interchangeability* was shown, substantial progress towards *interoperability* was not demonstrated.

Here we make a first step toward deeper component-based integration of quantum chemistry packages. In this work we develop a set of standard interfaces and data structures for evaluation of molecular integrals, and we demonstrate a component implementation of this design. Molecular integral evaluation is a natural low-level capability to share through components because it is the fundamental subproblem of all traditional quantum chemistry computations. As quantum chemical studies grow in sophistication, advanced capabilities, such as explicit electron correlation[?] and the inclusion of relativistic effects,[19–21] introduce the need for new types of integrals. The integral facilities available within each individual quantum chemistry package typically lack these advanced features, limiting the range of methods which are available to users of the package. Because writing efficient code for computing a new type of molecular integral requires significant development effort, it is natural to share the integral facilities as components. The obvious benefit of sharing integral capabilities between various packages is the ability to implement new theoretical methods very rapidly. For instance, in an early application of the work described here, combining integral capabilities from multiple packages allowed the latest explicitly correlated methods to take into account scalar relativistic effects.[22] While the scientific benefits of sharing low-level capabilities such as molecular integrals capabilities are strong, the potential performance impact of the component interface and data layout can grow beyond acceptable limits. Thus, this work serves as an important first test of the performance of CCA approaches when deeply integrated within performance-critical sections of code. Although we found the performance overhead of CCA approaches for the integral evaluation to be acceptable, good performance does require common standards for data format.

2 The Common Component Architecture

Some of the authors have previously described in detail[16,17] the CCA model in the context of quantum chemistry applications. We will only briefly summarize it here.

Components[23] are similar to objects in that they implement some functionality and provide an interface for using it. A programmer composes applications out of objects by writing computer code which instantiates objects and combines their functions. Although such applications can vary some of their functionality at runtime, via polymorphism, the full functionality of the application is determined once and for all at compile-time, i.e., *statically*.

In contrast, components conform to a particular environment specification which allows composition of applications at runtime, i.e., *dynamically*. The runtime environment which makes such composition possible is the component *framework*. The framework provides a simple (scripting or graphical) interface which allows each end user to compose highly customized software from plug-and-play components. The CCA specification[3–5] has been developed specifically to meet the requirements of high-performance scientific codes, stressing high performance and respecting parallel execution.

The Babel tool[11,12] is used within the CCA community to compose applications of components written in different languages. While not strictly required by the CCA specification, all components described herein utilize Babel. Babel is a code generator which provides implementation stubs and glue code in a collection of languages, based on interface definitions provided using the Scientific Interface Definition Language (SIDL). Babel/SIDL implements a set of fundamental data types, including complex and array types, and an object-oriented programming model for Fortran 77, Fortran 90, C, C++, Python, and Java. A SIDL *interface* declares methods based on these data types, and a SIDL *class* implements one or more interfaces. SIDL classes may be implemented in any supported language and may be used by applications written in any other supported language.

A CCA *component* in Babel is a SIDL class that implements the `Component` interface defined by the CCA specification and one or more programmer-defined interfaces. The `Component` interface simply defines the `setServices()` method, which is used by the framework to provide a `Services` object to the component during its instantiation. The `Services` object handles the interaction between the framework and the component. Its most important role is to inform the framework of interface implementations that the component either provides or requires. As illustrated in Figure 1, the component exposes a *provides port*, which specifies an abstract interface and provides an implementation of the interface, to the framework using the `addProvidesPort()` services method. A component can also request a *uses port* (an interface implementation) from the framework using the `registerUsesPort()` services method. Following the user's directives, the framework builds a component application by instantiating components and connecting uses ports with provides ports. Ideally, any well written software can be packaged into a component by adding a thin wrapper layer which handles interaction with the framework. Just like the object model, the component model tends to break down as deeper levels of integration expose implementation details at interfaces and lead to the propagation of implicit dependencies throughout the application.

3 Integral Component Interfaces

Here we overview the component interfaces which we have developed. A detailed description of the interfaces is found in Appendix A.

Figure 2 illustrates the component integral evaluation architecture. The key abstractions represented by the component interfaces are the integral factory (`IntegralFactoryInterface`) and integral evaluator (`IntegralEvaluatorInterface`). Integral evaluator objects compute molecular integrals. Currently, we specify integral evaluator interfaces for computing integrals involving one, two, three, and four Gaussian centers (`IntegralEvaluator1Interface`, etc.). Of course, integral evaluator interfaces can be easily extended to allow more centers. Each integral evaluator provides a `compute()` method which, given a shell multiplet, prompts the computation of the integrals.

In contrast to the use of the evaluator objects, their initialization can be very complicated, as it depends on the implementation details of the particular evaluator. The purpose of the integral factory is to hide the complexity of the initialization of integral evaluators. To create an integral evaluator, the factory must at a minimum receive the molecular basis set data. However, many integral types require the specification of additional data: derivative information, origin of the reference frame for dipole integrals, or exponents for Gaussian geminals, to name a few. The purpose of an integral descriptor (`IntegralDescriptorInterface`) object is to encapsulate such data. A set of descriptors must be provided to the factory to produce corresponding integral evaluators.

The only implementation detail which had to be reflected in the interface design was how to return buffers of computed integrals to client codes. Evaluator-allocated buffers are filled when the `compute()` method is called on an integral evaluator. In monolithic codes using pass-by-reference (Fortran family) or pointer-aware (C family) languages, buffer references or pointers are typically passed to the integrals client code, giving both integral evaluators and clients fast access to the buffer. Babel supports an `opaque` type, allowing integral evaluators to return the buffer pointer from a call to `get_buffer()`. Since non-pointer-aware languages (Java and Python) are supported by Babel and may be useful for prototyping, the interface supports an alternative to passing pointers. An additional compute method, `compute_array()`, is specified which returns a SIDL array with the contents of the integral buffer. While the SIDL array can be created without copying from the evaluator-allocated buffer, wrapper code for integrals clients will need to copy each SIDL array into the client's native buffers unless the client code is modified to operate on SIDL arrays. Both the `compute()` and `compute_array()` interfaces have been implemented and the performance impact of this copy operation will be examined in the following section.

4 Implementation and Benchmarking

The integrals interfaces we have developed have been implemented in the Massively Parallel Quantum Chemistry (MPQC) package.[24–26] This is currently the only implementation of CCA integral evaluators, though NWChem and GAMESS implementations are in progress. While the capabilities this adds to the MPQC package are minimal (the native integral packages can now be mixed), other packages may take advantage of these evaluators, and performance benchmarks will be valuable for evaluation of our approach and guidance in further development. Within the native MPQC code base, two integrals packages are supported. The IntV3 package is distributed with MPQC and performs integral evaluations needed for energies and gradients of conventional quantum methods. An additional package, Cints (based on the Libint[27,28] package), performs integral evaluations required for explicitly correlated methods in addition to those required for conventional methods. Both MPQC and Libint are open source, distributed under the GNU General Public License and Library GNU General Public License.

While the integral evaluation interfaces provide a framework in which to implement evaluator components, additional specifications must be imposed on the layout of data which is passed through these interfaces. We propose standards for integral data in Appendix B. The layout of integrals within buffers is of particular importance. While the Cints buffer layout does conform to the proposed CCA standard, the IntV3 buffer layout does not. Codes cannot be expected to flexibly handle different buffer layouts, and translation costs are significant. The CCA IntV3 integral evaluator includes the option to translate integral buffers into the standard CCA format.

As a general principle, larger amounts of time spent in MPQC's internal integral routines offset the overhead associated with the CCA interface layer, which adds several function calls and language interoperability code to each `compute()` call. Larger basis sets result in larger average buffer sizes, requiring more work per buffer computation and decreasing overhead. The bounds computations used in integral screening make up larger proportions of the computation as molecule size increases and more buffers are screened out. These bounds computations require a comparatively small amount of work per call. Thus, smaller molecules tend to have lower overhead. Similarly, gradient computations require more work per buffer than energy computations and tend to have reduced overhead. The test cases in the following tables are arranged in a manner which highlights these trends.

Tables I-A, I-B and I-C report average wall clock times for a number of energy and gradient calculations using either the native MPQC integral interfaces or the CCA component interfaces. Since Cints derivative integrals have not been exposed within MPQC, these calculations were performed using the IntV3 integrals package. The IntV3 native buffer layout is used for these calculations, thus yielding a measure of the performance of the CCA integral interfaces when integral reordering is not required, the highest performance case. The CCA overheads for Hartree-Fock calculations in optimum, small molecule, big basis set cases, exemplified by the water calculations in Table I-A, are under 1% and entirely insignificant. At the other end of the spectrum for Hartree-Fock methods, larger molecules with small basis sets can have significant overheads. In the benchmark suite for this study, the worst CCA overhead observed for the Hartree-Fock method was the isoprene/6-311++G** energy calculation, with an overhead of 8.2%. Correlated calculations, such as the MP2 calculations in Table I-B, generally require greater computational effort subsequent to atomic orbital integral evaluation and, thus, experience moderately lower CCA overheads which are below 5%. Density functional approaches, as surveyed in Table I-C, expend significant computational effort in numerical integration which is not currently performed through a CCA interface. Consequently, overheads for the B3LYP calculations in Table I-C are uniformly low, with the greatest overhead of 3.1% seen in the water energy calculation.

Tables II-A, II-B and II-C compare average wall clock times for test cases using CCA interfaces and IntV3 integrals, comparing the use of the IntV3 native buffer layout with the CCA buffer layout (including buffer translation overhead). This overhead, which is in addition to the CCA interface overhead, is substantial, with Hartree-Fock reorder overheads as high 11.8% and MP2 reorder overheads as high as 10.3%. As in the CCA overheads, density functional approaches have moderately lower overheads for reordering with a maximum observed overhead of 3.8% for the isoprene/6-311++G** energy evaluation. Deeply nested loops, inefficient memory access patterns and logic to handle contractions, derivatives and differing angular types are unavoidable and result in a substantially expensive reorder algorithm. Maintaining good performance with low-level interfaces requires minimizing such translation costs and, while such overhead is undoubtedly unavoidable when using legacy codes, we strongly advocate standards adoption for any new development efforts.

As discussed previously, the use of SIDL arrays rather than opaque types to access integral buffers, while obviously reducing performance, is necessary to support Java and Python. To assess the impact of the added client-side copying from SIDL to native buffers, Table III reports timings for calculations comparing the opaque and SIDL array approaches. The overhead due to using SIDL arrays for integral buffers follows the same pattern as CCA interface and integral reordering overheads. Hartree-Fock methods again show the greatest performance loss, with the difficult isoprene/cc-pVDZ energy incurring a 14.4% overhead when using SIDL arrays. Reduced overhead was seen for MP2 and, more significantly, density functional approaches, with a maximum observed overhead of 5.7% for the isoprene/cc-pVDZ gradient. As before, the water/aug-cc-pV5Z cases exemplify the high performance which can be expected for small molecule calculations using large basis sets, with a maximum observed overhead of 2.4% for the Hartree-Fock energy.

Since integral packages are now interchangeable through the CCA interfaces, the source package for each of a calculation's integral types can be selected individually. New packages implementing advanced integral types no longer need to duplicate standard integral capabilities, as these are available from a growing number of CCA implementations based on mature integral packages. Table IV provides timings for several MP2-R12/A' test cases demonstrating this flexibility. For the first set of tests, Cints integrals were used throughout. For comparison, the second set uses Cints for two-electron integrals while IntV3 is used to provide overlap and core Hamiltonian integrals. Though reordering of the IntV3 buffers was required for the second set of calculations, very little overhead is observed since the cost of one-electron integrals is minor. These calculations also highlight opportunities for quality of service improvements which the CCA architecture enables. With a large set of interchangeable integral evaluator implementations, it is possible to include components for automated selection of the most appropriate implementation for a given integral type, calculation type and hardware environment.

5 Embedded Frameworks

The components previously developed for geometry optimization[16,17] encapsulted the high-level functionalities of various domain-specific packages. There was no existing support for constructing and configuring applications based on the disparate packages, making an entirely component-based architecture composed and configured via the framework interface a natural choice. While the flexibility afforded by direct interaction with the framework is valuable, the assumption that end-users will familiarize themselves with the concepts and skills necessary to properly configure component applications may be too severe. The development of a standard input format that generically supports quantum chemistry packages was recognized as both a unique opportunity and a daunting challenge of this design, and has not, thusfar, been attempted.

Shifting some low-level functionality to components more fully realizes the potential of component technology to facilitate interoperable, rather than just interchangeable, chemistry packages, and suggests the possibility of applications which only utilize components for specific tasks. The CCA specification and Ccaffeine framework do support embedding of framework functionality in stand-alone codes, allowing mixed legacy/component application architectures. Surely, each chemistry package supports configuration of computations, and this configuration capability can be easily extended to support accessing low-level functionality provided by components. Embedding component composition and configuration inside legacy codes allows package developers to expose to end-users only those features useful in a particular context. The barrier for end-users to adopt component technology is substantially reduced; the stand-alone package is run as before with a small number of extra parameters introduced to calculation inputs. An embedded framework environment has been added along with the integrals component client and server code that we have added to the MPQC package, allowing MPQC chemistry models to access integrals via components in both exclusively component and mixed legacy/component modes. The adoption of integrals components by MPQC users is now trivial.

6 Conclusions

Through the design and implementation of an extendable interface for molecular integral evaluation in quantum chemistry, we have demonstrated the efficacy of exposing low-level software capabilities using component approaches. For integral evaluation, calls through component interfaces easily number in the millions and involve several extra function calls and a fair amount of language interoperability code, yet overheads are reasonable. For calculations on small molecules using large basis sets, interface overheads below 1% can be expected. For larger molecules and smaller basis sets, interface overheads are clearly more significant but are acceptable, with all benchmark values falling within 10%. While overheads in the 10% range are likely large enough to discourage the use of these interfaces when high performance native implementations are available, the ability to share integral types and rapidly implement new approaches ensures the usefulness of integral components.

When low-level software functionality is shared between packages, adapting arbitrary implementation details, such as integral buffer ordering, to a common standard can cause substantial overhead which overshadows the costs of an added interface layer. In the case of integral buffer ordering, our benchmarks show overheads as high as 12%, in addition to the interface overhead due to the CCA layer. While such significant overheads will often be unavoidable when adapting legacy codes for component implementations, the development and adoption of standards in new work is critical to reducing such inefficiencies in the long term, allowing for a high performance community code base.

As our efforts in developing a chemistry component toolkit have progressed, it has become evident that the CCA approach is about much more than components. The true strength of the

CCA is as a comprehensive approach to scientific software engineering. Holding as much or more importance than component concepts are the approaches to interface and data standards, language and package interoperability, and collaborative, community-based development which have naturally evolved along with component standards and middleware. In our view, the component concept has functioned as a catalyst which has focused a large community of computational and computer scientists on solving the problems of large scale, collaborative, scientific software development. While improving usability and maintenance of this increasingly complex chemistry software project remains a challenge, this work is significant progress in developing an interoperable code base for quantum chemistry.

Acknowledgements

The authors acknowledge the contributions of Dr. David E. Bernholdt and Dr. Ricky A. Kendall during the initial interface design. This work has been supported in part by the U.S. Department of Energy's Scientific Discovery through Advanced Computing (SciDAC) initiative,[29] through the Center for Component Technology for Terascale Simulation Software, of which Oak Ridge, Pacific Northwest, and Sandia National Laboratories are members. This research was performed in part using the Molecular Science Computing Facility (MSCF) in the William R. Wiley Environmental Laboratory at the Pacific Northwest National Laboratory (PNNL). The MSCF is funded by the Office of Biological and Environmental Research in the U.S. Department of Energy. Research of EFV was supported in part by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the U. S. Department of Energy under Contract No. DE-AC05-00OR22725; and by start-up funding from the Department of Chemistry at Virginia Tech. PNNL is operated by Battelle for the U.S. Department of Energy under contract DE-AC06-76RLO 1830. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the U.S. Department of Energy under contract DE-AC04-94AL85000.

Calculation Details

The codes implementing CCA integral interfaces are currently under development and will be available in forthcoming releases. Source code for the cca-chem-generic package,[17] which provides interface definitions and some implementations which are generally useful, was a snapshot of the babel-1-0 branch as of 11/3/2006. MPQC[24–26] source code was a snapshot of MPQC's babel-1-0 branch as of 11/7/2006. Babel[11,12] source code was a snapshot of Babel's 1.1 development branch as of 12/21/2006. Source code for the cca-tools package[5] was the 0.6.1_rc2 release, with slight modifications to allow building against Babel 1.1. Ccaffeine,[6] one of several frameworks[6–10] that comply with the CCA specification, was used. All codes were built using the gcc 3.4.3 compiler with the default x86_64 processor target and -O2 optimization. Benchmarks were performed in single process, single thread mode on Intel Xeon 5160 CPU's (Core 2 / Woodcrest architecture) clocked at 3.00 GHz, running Red Hat Enterprise Linux AS release 4.

References

[1] van der Steen, A. J. Overview of Recent Supercomputers; http://www.phys.uu.nl/˜euroben/reports/web05a/overview.html.

[2] Földesy, P. in Proceedings. 2004 IEEE International Joint Conference on Neural Networks; IEEE Press: Santa Fe, 2004.

[3] Armstrong, R.; Gannon, D.; Geist, A.; Keahey, K.; Kohn, S.; McInnes, L.; Parker, S.; Smolinski, B. in Proceedings. The Eighth International Symposium on High Performance Distributed Computing; IEEE Press: Santa Fe, 1999.

[4] Bernholdt, D. E.; Allan, B. A.; Armstrong, R.; Bertrand, F.; Chiu, K.; Dahlgren, T. L.; Damevski, K.; Elwasif, W. R.; Epperly, T. G. W.; Govindaraju, M.; Katz, D. S.; Kohl, J. A.; Krishnan, M.; Kumfert, G.; Larson, J. W.; Lefantzi, S.; Lewis, M. J.; Malony, A. D.; McInnes, L. C.; Nieplocha, J.; Norris, B.; Parker, S. G.; Ray, J.; Shende, S.; Windus, T. L.; Zhou, S. Intl J High-Perf Computing Appl 2006, 20, 163-202.

[5] CCA Forum; CCA Forum homepage; http://www.cca-forum.org/.

[6] Allan, B. A.; Armstrong, R. C.; Wolfe, A. P.; Ray, J.; Bernholdt, D. E.; Kohl, J. A. Concurrency and Computation: Practice and Experience 2002, 14, 1-23.

[7] Govindaraju, M.; Krishnan, S.; Chiu, K.; Slominski, A.; Gannon, D.; Bramley, R. in 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid; IEEE Press: Santa Fe, 2003.

[8] Zhang, K.; Damevski, K.; Venkatachalapathy, V.; Parker, S. in Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004); IEEE Press: Santa Fe, 2004, to appear.

[9] Grimshaw, A.; Ferrari, A.; Knabe, F.; Humphrey, M. Computer 1999, 32, 29-37.

[10] Lewis, M.; Ferrari, A.; Humphrey, M.; Karpovich, J.; Morgan, M.; Natrajan, A.; Nguyen-Tuong, A.; Wasson, G.; Grimshaw, A. J Parallel and Distributed Comp 2003, 63, 525-538.

[11] Dahlgren, T.; Epperly, T.; Kumfert, G. Babel User's Guide, version 0.9.0, 2004.

[12] Lawrence Livermore National Laboratory; Babel homepage; http://www.llnl.gov/CASC/components/babel.html.

[13] 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. in Solutions of PDEs on Parallel Computers; Springer-Verlag: New York, 2005; Chapter Parallel PDE-Based Simulations Using the Common Component Architecture.

[14] Lefantzi, S.; Ray, J.; Najm, H. N. in Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003); IEEE Press: Santa Fe, 2003.

[15] Lefantzi, S.; Ray, J.; Kennedy, C. A.; Najm, H. N. Prog Comput Fluid Dy 2005, 5, 298-315.

[16] 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.

[17] CCA Quantum Chemistry Working Group; Homepage; http://www.cca-forum.org/˜cca-chem.

[18] Krishnan, M.; Alexeev, Y.; Windus, T. L.; Nieplocha, J. in Proceedings of the 2005 ACM/IEEE Conference on Supercomputing; IEEE Computer Society: Washington, D.C., 2005.

[19] Douglas, M.; Kroll, N. M. Ann Phys 1974, 82, 89.

[20] Hess, B. A. Phys Rev A 1986, 33, 3742.

[21] Pacios, I. F.; Christiansen, P. A. J Chem Phys 1985, 82, 2664.

[22] 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.

[23] Szyperski, C. Component Software: Beyond Object-Oriented Programming; Addison-Wesley: New York, 2002; Chapter 4.

[24] Janssen, C. L.; Nielsen, I. M. B.; Colvin, M. E. in Encyclopedia of Computational Chemistry; John Wiley & Sons: Chichester, UK, 1998; Chapter Parallel Processing for ab Initio Quantum Mechanical Methods.

[25] Sandia National Laboratories; MPQC homepage; http://aros.ca.sandia.gov/˜cljanss/mpqc.

[26] Janssen, C.; Seidl, E.; Colvin, M. in ACS Symposium Series, Parallel Computing in Computational Chemistry, Vol. 592; American Chemical Society: Washington, D.C., 1995.

[27] Department of Chemistry, Virginia Tech; Valeev, E. F. Libint library; http://www.chem.vt.edu/chem-dept/valeev/libint/.

[28] Valeev, E. F.; Janssen, C. L. J Chem Phys 2004, 121, 1214.

[29] U. S. Dept. of Energy; SciDAC Initiative homepage; http://www.osti.gov/scidac/.

[30] Schlegel, H. B.; Frisch, M. J. Int J Quant Chem 1995, 54, 83-87.

Table I-A. Average wall times (seconds) for Hartree-Fock calculations performed using native and CCA integral interfaces. The IntV3 integral package was used with native buffer layouts throughout. Calculations were repeated three times.

		Native Buffer Layout		
Test Case	Basis Set	Native Interface Wall Time	CCA Interface Wall Time	CCA Overhead
HF Energy				
isoprene	6-311++G**	184	199	8.2 %
	cc-pVDZ	33.5	36.1	7.8 %
	cc-pVTZ	904	941	4.1 %
phosphoserine	cc-pVDZ	254	265	4.3 %
aniline	cc-pVTZ	915	947	3.5 %
water	aug-cc-pV5Z	950	956	0.6 %
HF Gradient				
isoprene	6-311++G**	364	389	6.9 %
	cc-pVDZ	92.4	97.8	5.8 %
	cc-pVTZ	2236	2316	3.6 %
phosphoserine	cc-pVDZ	669	694	3.7 %
aniline	cc-pVTZ	1790	1845	3.1 %
water	aug-cc-pV5Z	1757	1769	0.7 %

Table I-B. Average wall times (seconds) for MP2 calculations performed using native and CCA integral interfaces. The IntV3 integral package was used with native buffer layouts throughout. Calculations were repeated three times.

Test Case	Basis Set	Native Buffer Layout Native Interface Wall Time	 CCA Interface Wall Time	CCA Overhead
MP2 Energy				
isoprene	6-311++G**	117	120	2.6 %
	cc-pVDZ	42.4	43.3	2.1 %
	cc-pVTZ	581	591	1.7 %
phosphoserine	cc-pVDZ	585	591	0.2 %
aniline	cc-pVTZ	1168	1175	0.6 %
water	aug-cc-pV5Z	258	260	0.8 %
MP2 Gradient				
isoprene	6-311++G**	688	717	4.2 %
	cc-pVDZ	235	244	3.8 %
	cc-pVTZ	4182	4277	2.3 %
phosphoserine	cc-pVDZ	2993	3034	1.4 %
aniline	cc-pVTZ	5883	5960	1.3 %
water	aug-cc-pV5Z	3838	3820	negligible

Table I-C. Average wall times (seconds) for B3LYP calculations performed using native and CCA integral interfaces. The IntV3 integral package was used with native buffer layouts throughout. Calculations were repeated three times.

		Native Buffer Layout		
Test Case	Basis Set	Native Interface Wall Time	CCA Interface Wall Time	CCA Overhead
B3LYP Energy				
isoprene	6-311++G**	645	659	2.2 %
	cc-pVDZ	208	210	1.0 %
	cc-pVTZ	1852	1891	2.1 %
phosphoserine	6-311G**	968	984	1.7 %
aniline	6-311++G**	956	966	1.0 %
water	6-311++G(3df,3pd)	12.8	13.2	3.1 %
B3LYP Gradient				
isoprene	6-311++G**	1602	1622	1.2 %
	cc-pVDZ	723	725	0.3 %
	cc-pVTZ	4411	4481	1.6 %
phosphoserine	6-311G**	2288	2314	1.1 %
aniline	6-311++G**	1987	2007	1.0 %
water	6-311++G(3df,3pd)	46.6	47.1	1.1 %

Table II-A. Average wall times (seconds) for Hartree-Fock calculations performed using CCA integral interfaces. IntV3 integals are used throughout, comparing native and CCA buffer orderings. Calculations were repeated three times.

Test Case	Basis Set	CCA Interface Native Buffer Layout Wall Time	CCA Interface CCA Buffer Layout Wall Time	Reorder Overhead
HF Energy				
isoprene	6-311++G**	199	220	10.6 %
	cc-pVDZ	36.1	39.6	9.7 %
	cc-pVTZ	941	1002	6.5 %
phosphoserine	cc-pVDZ	265	280	5.7 %
aniline	cc-pVTZ	947	999	5.5 %
water	aug-cc-pV5Z	956	982	2.7 %
HF Gradient				
isoprene	6-311++G**	389	435	11.8 %
	cc-pVDZ	97.8	105.7	8.1 %
	cc-pVTZ	2316	2474	6.8 %
phosphoserine	cc-pVDZ	694	728	4.9 %
aniline	cc-pVTZ	1845	1947	5.5 %
water	aug-cc-pV5Z	1769	1830	3.4 %

Table II-B. Average wall times (seconds) for MP2 calculations performed using CCA integral interfaces. IntV3 integals are used throughout, comparing native and CCA buffer orderings. Calculations were repeated three times.

		CCA Interface		
		Native Buffer Layout	CCA Buffer Layout	
Test Case	Basis Set	Wall Time	Wall Time	Reorder Overhead
MP2 Energy				
isoprene	6-311++G**	120	127	5.8 %
	cc-pVDZ	43.3	44.6	3.0 %
	cc-pVTZ	591	610	3.2 %
phosphoserine	cc-pVDZ	591	595	0.7 %
aniline	cc-pVTZ	1175	1255	6.8 %
water	aug-cc-pV5Z	260	265	1.9 %
MP2 Gradient				
isoprene	6-311++G**	717	791	10.3 %
	cc-pVDZ	244	258	5.7 %
	cc-pVTZ	4277	4547	6.3 %
phosphoserine	cc-pVDZ	3034	3113	2.6 %
aniline	cc-pVTZ	5960	6276	5.3 %
water	aug-cc-pV5Z	3820	3956	3.6 %

Table II-C. Average wall times (seconds) for B3LYP calculations performed using CCA integral interfaces. IntV3 integals are used throughout, comparing native and CCA buffer orderings. Calculations were repeated three times.

		CCA Interface		
Test Case	Basis Set	Native Buffer Layout Wall Time	CCA Buffer Layout Wall Time	Reorder Overhead
B3LYP Energy				
isoprene	6-311++G**	659	684	3.8 %
	cc-pVDZ	210	214	1.9 %
	cc-pVTZ	1891	1962	3.8 %
phosphoserine	6-311G**	984	1019	3.6 %
aniline	6-311++G**	966	991	2.6 %
water	6-311++G(3df,3pd)	13.2	13.4	1.5 %
B3LYP Gradient				
isoprene	6-311++G**	1622	1667	2.8 %
	cc-pVDZ	725	736	1.5 %
	cc-pVTZ	4481	4628	3.3 %
phosphoserine	6-311G**	2314	2387	3.2 %
aniline	6-311++G**	2007	2040	1.6 %
water	6-311++G(3df,3pd)	47.1	47.6	1.1 %

Table III. Average wall times (seconds) for calculations performed using CCA integral interfaces. Intv3 integals with native buffer layouts are used throughout, comparing opaque and SIDL array buffer access. Calculations were repeated three times.

Test Case	Basis Set	Opaque Access Wall Time	SIDL Array Access Wall Time	SIDL Array Access Overhead
HF Energy				
isoprene	cc-pVDZ	36.1	41.3	14.4 %
water	aug-cc-pV5Z	956	979	2.4 %
HF Gradient				
isoprene	cc-pVDZ	97.8	107	9.4 %
water	aug-cc-pV5Z	1769	1781	0.7 %
MP2 Energy				
isoprene	cc-pVDZ	43.3	45.5	5.1 %
water	aug-cc-pV5Z	260	265	1.9 %
MP2 Gradient				
isoprene	cc-pVDZ	244	258	5.7 %
water	aug-cc-pV5Z	3820	3840	0.5 %
B3LYP Energy				
isoprene	6-311++G**	659	694	5.3 %
water	6-311++G(3df,3pd)	13.2	13.3	0.8 %
B3LYP Gradient				
isoprene	6-311++G**	1622	1673	3.1 %
water	6-311++G(3df,3pd)	47.1	47.3	0.4 %

Table IV. Average wall times (seconds) for calculations performed using CCA integral interfaces. Calculations using Cints integals throughout and using Cints two-electron integrals along with IntV3 one-electron integrals are compared. CCA buffer layouts (identical to Cints buffer layouts) were used throughout. Calculations were repeated three times.

Test Case	Basis Set	Cints Wall Time	Cints/IntV3 Wall Time	IntV3 Overhead
MP2-R12 Energy				
isoprene	cc-pVDZ	245.9	245.9	0.0 %
water	aug-cc-pV5Z	1549	1551	0.1 %

Figure Captions:

Figure 1. A schematic representation of ports in the CCA approach. `Component A` provides a port, `PortA`, for which `Component B` gains access.

Figure 2. A schematic representation of the component integral evaluator architecture.

Figure 1.

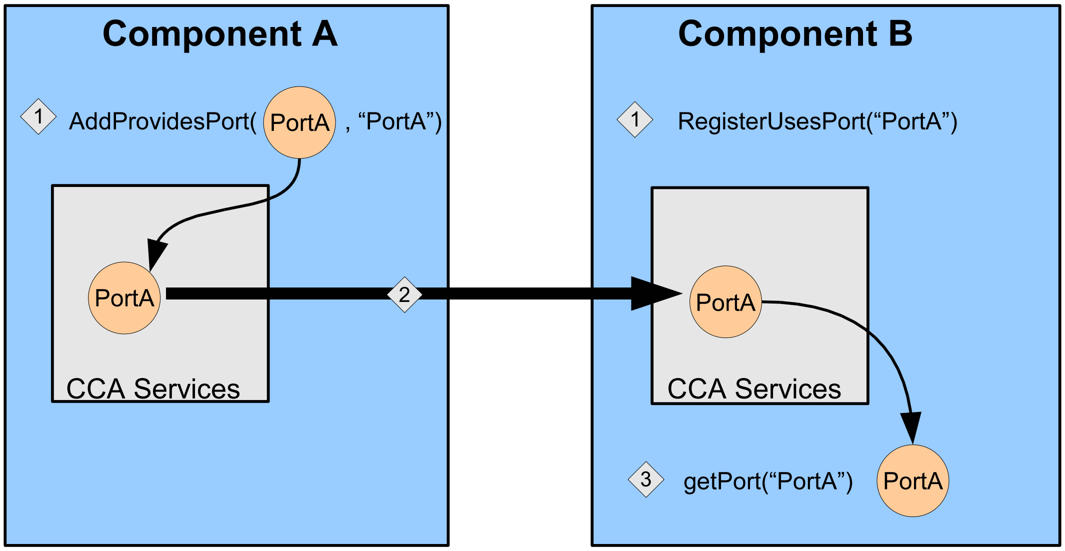
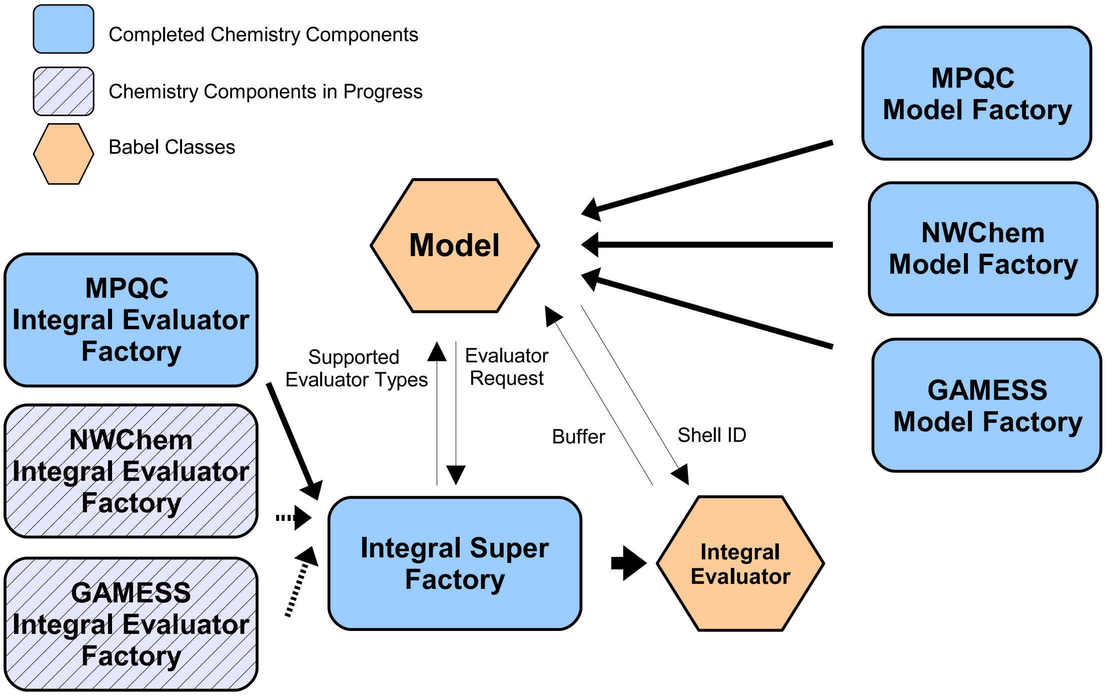

Component A
1
AddProvidesPort(PortA , “PortA”)
PortA
CCA Services
2
Component B
1
RegisterUsesPort(“PortA”)
PortA
CCA Services
3
getPort(“PortA”)
PortA

Figure 2.


Completed Chemistry Components
Chemistry Components in Progress
Babel Classes
MPQC Model Factory
NWChem Model Factory
GAMESS Model Factory
Model
MPQC Integral Evaluator Factory
NWChem Integral Evaluator Factory
GAMESS Integral Evaluator Factory
Supported Evaluator Types
Evaluator Request
Buffer
Shell ID
Integral Super Factory
Integral Evaluator

Appendix A: Integral Evaluation Interfaces

Interfaces concerned with integral evaluation over Gaussian basis functions are described in the following sections. Babel's Scientific Interface Description Language (SIDL)[11] is a neutral choice for this description. The SIDL code for the following interfaces resides in the `Chemistry.QC.GaussianBasis` package, a subsection of the `chemistry.sidl` file distributed with the `cca-chem-generic` software.

For convenience, an enumeration for function angular types is provided.

enum AngularType { **CARTESIAN, SPHERICAL, MIXED** }

Codes which utilize integral evaluator components must pass Gaussian basis set data to the evaluators. This task is performed by copying basis set data into class implementations of the `ShellInterface`, `AtomicInterface`, and `MolecularInterface` and passing the `MolecularInterface` object to the integral evaluators (a `MolecularInterface` *object* is any SIDL class which implements the `MolecularInterface`). A `MolecularInterface` object, representing a molecular basis set, contains a set of `AtomicInterface` objects, each of which contains a set of `ShellInterface` objects.

ShellInterface

int get_n_contraction()
Get the number of contractions in the shell.
Returns:
number of contractions

int get_n_primitive()
Get the number of primitives in the shell.
Returns:
number of primitives

double get_contraction_coef(in int connum, in int expnum)
Get the coefficient for an unnormalized primitive.
Returns:
contraction coefficient
Parameters:
connum *contraction number*
expnum *primitive number*

double get_exponent(in int expnum)
Get the exponent for a primitive.

Returns:
exponent

Parameters:
expnum *primitive number*

int get_angular_momentum(in int connum)
Get the angular momentum for a single contraction.

Returns:
angular momentum value

Parameters:
connum *contraction number*

int get_max_angular_momentum()
Get the max angular momentum, considering all contractions in the shell.

Returns:
maximum angular momentum value

AngularType get_contraction_angular_type(in int connum)
Get the angular type for a single contraction.

Returns:
angular type

Parameters:
connum *contraction number*

AngularType get_angular_type()
Get the shell angular type.

Returns:
angular type

void print_shell()
Print the shell data.

AtomicInterface

string get_name()
Get the canonical basis set name.

Returns:
canonical name

int get_n_basis()
Get the number of basis functions.

Returns:
number of functions

int get_n_shell()
Get the number of shells.
    Returns:
        number of shells

int get_max_angular_momentum()
Get the max angular momentum for any shell on the atom.
    Returns:
        max angular momentum value

AngularType get_angular_type()
Get the angular type for the atom.
    Returns:
        angular type

ShellInterface get_shell(in int shellnum)
Get a gaussian shell.
    Returns:
        object implementing **ShellInterface**
    Parameters:
        shellnum *shell number*

void print_atomic()
Print the atomic basis data.

MolecularInterface

string get_label()
Get the user specified name.
    Returns:
        name

long get_n_basis()
Get the number of basis functions.
    Returns:
        number of functions

long get_n_shell()
Get the number of shells.
    Returns:
        number of shells

int get_max_angular_momentum()
Get the max angular momentum for any contraction in the basis set.

Returns:

max angular momentum value

AngularType get_angular_type()
Get the angular type.

Returns:

angular type

AtomicInterface get_atomic(in long atomnum)
Get an atomic basis set.

Returns:

object implementing the **AtomicInterface**

Parameters:

atomnum *atom number*

MoleculeInterface get_molecule()
Get the molecule.

Returns:

object implementing the **MoleculeInterface**

void print_molecular()
Print the molecular basis data.

Many integral types require data in addition to integral buffers to be shared between servers and clients. A number of data container utility interfaces serve this function, and this collection will grow as client/server components with additional capabilities are implemented. The first such utility interface we describe is the `DerivCentersInterface`, used to specify details for nuclear derivative calculations. Code utilizing an integral evaluator must pass a `DerivCentersInterface` object to the evaluator. For derivatives with operators which are independent of nuclear coordinates, translational invariance allows derivatives with respect to one center to be ommitted, and details about omitted centers are shared through this interface. For derivatives of operators which are dependent upon nuclear coordinates, derivatives with respect to every atom must be taken, and the derivative atom information is shared using the `set_deriv_atom()` and `get_deriv_atom()` methods. The *segment* number, defined as the number of basic buffer segments a buffer is composed of, is also shared through this interface. As an example of the buffer segment concept, a dipole integral buffer consists of one buffer segement for each of x, y, and z, yielding a segment number of 3.

DerivCentersInterface

void clear()

Clear the list of centers.

Returns:

list of centers

void add_center(in long center, in long atom)

Add a center for which derivatives will be computed.

Parameters:

center *center number (between 0 and 3 inclusive)*

atom *atom number corresponding to center*

void add_omitted(in long center, in long atom)

Add a center for which derivatives will not be computed.

Parameters:

center *center number (between 0 and 3 inclusive)*

atom *atom number corresponding to center*

long n()

Returns the number of centers for which derivatives will be computed.

Returns:

number of centers

long center(in long i)

Returns center number.

Returns:

center number (between 0 and 3 inclusive)

Parameters:

i *computed center index (between 0 and n()-1 inclusive)*

long atom(in long i)

Returns atom number.

Returns:

atom number

Parameters:

i *computed center index (between 0 and n()-1 inclusive)*

long omitted_center()

Returns the omitted center number.

Returns:

omitted center number

int has_omitted_center()
Returns 1 if there is an omitted center.
Returns:
1 (true) or 0 (false)

long omitted_atom()
Returns atom that is omitted from the integral buffer.
Returns:
omitted atom number

void set_deriv_atom(in int deriv_atom)
Set the atom a derivative is taken with respect to.
Parameters:
deriv_atom *atom number*

int get_deriv_atom()
Get the atom a derivative is taken with respect to.
Returns:
atom number

Currently, the only additional data container utility interface is the `DipoleDataInterface` which provides origin information for dipole and quadrupole integrals.

DipoleDataInterface

void set_origin(in array<double> origin)
Set the dipole origin.
Parameters:
origin *Cartesian coordinate array*

array<double> get_origin()
Get the dipole origin.
Returns:
Cartesian coordinate array

When an integral evaluator is requested, a composite of `IntegralDescrInterface` objects for the requested integral types is passed to the evaluator factory. This action provides both the list of required integral types and any additional data required. The object oriented features of Babel allow a collection of derived integral descriptors to be upcast to a collection of base descriptors which are passed through the `EvaluatorFactoryInterface`. The evaluator factory then checks descriptor types, performs any necessary downcasts and thereby obtains the auxilliary data. The `EvaluatorFactoryInterface` is thus generic and extendable for all possible integral types, requiring only the implementation of a new derived integral descriptor for types requiring additional data. We now describe the `IntegralDescrInterface`, `CompositeIntegralDescrInterface`, and `IntegralEvaluatorFactoryInterface`, along with an example derived descriptor interface, the `DipoleIntegralDescrInterface`.

IntegralDescrInterface

string get_type()
Get integral type.
  Returns:
    type

int get_n_segment()
Get number of segments.
  Returns:
    number of segments

void set_deriv_lvl(in int deriv)
Set derivative level.
  Parameters:
    deriv *derivative level*

int get_deriv_lvl()
Get derivative level.
  Returns:
    derivative level

void set_deriv_centers(in DerivCentersInterface dc)
Set derivative centers object.
  Parameters:
    dc *derivative centers object*

DerivCentersInterface get_deriv_centers()
Get derivative centers object.
  Returns:
    derivative centers object

DipoleIntegralDescrInterface

extends **IntegralDescrInterface**

void set_dipole_data(in DipoleDataInterface dipole_data)
Set the dipole data.
Parameters:
dipole_data *dipole data*

DipoleDataInterface get_dipole_data()
Get the dipole data.
Returns:
dipole data

CompositeIntegralDescrInterface

void add_descr(in IntegralDescrInterface desc)
Add an integral descriptor.
Parameters:
desc *integral descriptor*

int get_n_descr()
Get number of descriptors contained.
Returns:
number of descriptors

IntegralDescrInterface get_descr(in int n)
Get an integral descriptor.
Returns:
integral descriptor
Parameters:
n *descriptor index*

int is_contained(in IntegralDescrInterface desc)
Query if a descriptor with matching type and derivative level is contained.
Returns:
1 (true) or 0 (false)
Parameters:
desc *integral descriptor*

void clear()
Clear all descriptors.

IntegralEvaluatorFactoryInterface

string get_name()
Get factory name.
  Returns:
    name

CompositeIntegralDescrInterface get_descriptor()
Get composite of descriptors for supported integrals.
  Returns:
    composite integral descriptor

bool is_supported(in IntegralDescrInterface desc)
Query if a type and derivative level is supported.
  Returns:
    true or false
  Parameters:
    desc *integral descriptor*

void set_storage(in long storage)
Set storage that the factory is allowed to utilize.
  Parameters:
    storage *allowed storage in bytes*

IntegralEvaluator1Interface
get_evaluator1(
  in CompositeIntegralDescrInterface desc,
  in MolecularInterface bs1
)
Get a 1-center integral evaluator.
  Returns:
    1-center evaluator
  Parameters:
    desc *composite integral descriptor*
    bs1 *basis set for center 1*

IntegralEvaluator2Interface
get_evaluator2(
in CompositeIntegralDescrInterface desc,
in MolecularInterface bs1,
in MolecularInterface bs2
)

Get a 2-center integral evaluator.

Returns:

2-center evaluator

Parameters:

desc *composite integral descriptor*
bs1 *basis set for center 1*
bs2 *basis set for center 2*

IntegralEvaluator3Interface
get_evaluator3(
in CompositeIntegralDescrInterface desc,
in MolecularInterface bs1,
in MolecularInterface bs2,
in MolecularInterface bs3
)

Get a 3-center integral evaluator.

Returns:

3-center evaluator

Parameters:

desc *composite integral descriptor*
bs1 *basis set for center 1*
bs2 *basis set for center 2*
bs3 *basis set for center 3*

IntegralEvaluator4Interface
get_evaluator4(
in CompositeIntegralDescrInterface desc,
in MolecularInterface bs1,
in MolecularInterface bs2,
in MolecularInterface bs3,
in MolecularInterface bs4
)

Get a 4-center integral evaluator.

Returns:

4-center evaluator

Parameters:

desc *composite integral descriptor*
bs1 *basis set for center 1*
bs2 *basis set for center 2*
bs3 *basis set for center 3*
bs4 *basis set for center 4*

int finalize()

This should be called when the object is no longer needed. No other members may be called after finalize.

Returns:

0 on success

The `IntegralSuperFactoryInterface` provides a management layer for simplifying the use of multiple integral evaluator factories (following the Abstract Factory Pattern). Once client code provides rules for the routing of integral evaluator requests, the super factory acts as a single evaluator factory enveloping the capapbilities of all connected factories.

IntegralSuperFactoryInterface

extends **IntegralEvaluatorFactoryInterface**

array<string,1> add_uses_ports(in int n)
Add uses ports to component implementation.
Returns:
array of uses port names
Parameters:
n *number of additional uses ports*

void remove_port(in int portid)
Remove uses port.
Parameters:
portid *port index*

array<string,1> get_port_names()
Get uses port names.
Returns:
array of uses port names

array<string,1> get_factory_names()
Get attached evaluator factory names.
Returns:
array of factory names

array<CompositeIntegralDescrInterface,1> get_descriptors()
Get composite of available integral descriptors for each factory.
Returns:
array of composite integral descriptors

void set_default_subfactory(in string fac)
Set the default factory for integral evaluator requests.
Parameters:
fac *name of default factory*

void
set_subfactory_config(
  in array<string,1> types,
  in array<string,1> derivs,
  in array<string,1> facs
)
Configure which factory handles specific integral type requests.
  Parameters:
    types *array of integral types*
    derivs *array of derivative levels (integer or ”n” for wildcard)*
    facs *array of factory names*

The remaining interfaces specify the integral evaluator interfaces themselves. A base interface, `IntegralEvaluatorInterface`, is extended for one, two, three, and four-center integrals. We describe the two-center interface here; extension to other numbers of centers is obvious.

IntegralEvaluatorInterface

opaque get_buffer(in IntegralDescrInterface desc)
Get buffer pointer for given type.
  Returns:
    buffer pointer
  Parameters:
    desc *integral descriptor*

CompositeIntegralDescrInterface get_descriptor()
Get composite of descriptors for integral types supported.
  Returns:
    composite integral descriptor

int finalize()
This should be called when the object is no longer needed. No other members may be called after finalize.
  Returns:
    0 on success

IntegralEvaluator2Interface

void compute(in long shellnum1, in long shellnum2)
Compute all buffers for specified shell multiplet.
  Parameters:
    shellnum1 *shell 1 index*
    shellnum2 *shell 2 index*

array<double,1>
compute_array(
in string type,
in int deriv_lvl,
in long shellnum1,
in long shellnum2
)

Compute buffer for specified type, deriv level and shell multiplet, and return as SIDL array.

Returns:
SIDL buffer

Parameters:
type *integral type*
deriv_lvl *derivative level*
shellnum1 *shell 1 index*
shellnum2 *shell 2 index*

double compute_bounds(in long shellnum1, in long shellnum2)

Compute max integral bound.

Returns:
max integral bound for all computed types

Parameters:
shellnum1 *shell 1 index*
shellnum2 *shell 2 index*

array<double> compute_bounds_array(in long shellnum1, in long shellnum2);

Compute integral bounds for each computed type.

Returns:
SIDL array of integral bounds

Parameters:
shellnum1 *shell 1 index*
shellnum2 *shell 2 index*

Appendix B: Proposed Integral Standards

While the integral interfaces we have proposed define a set of function calls which may be used to obtain, initialize, and utilize molecular integral evaluators, standards must specified for implementational details, namely buffer layout and normalization conventions.

Buffer Layout

The most intuitive algorithm for the ordering of cartesian functions is proposed. Given angular momentum *l*, the cartesian functions $x^a y^b z^c$ are ordered as follows

starting with

$a = l$
$b = c = 0$

the next function is given by

```
$if(c < l - a)$   {
$b = b - 1$
$c = c + 1$
}

$else$   {
$a = a - 1$
$c = 0$
$b = l - a$
}
```

For example, a *d* shell is ordered

$$x^2, \;\; xy, \;\; xz, \;\; y^2, \;\; yz, \;\; z^2$$

For indexing within a cartesian shell multiplet buffer, the first center is treated as the most significant, with each subsequent center receiving less significance.

For a *pp* shell doublet the ordering is

$$\begin{array}{lll} < x|x > & < x|y > & < x|z > \\ < y|x > & < y|y > & < y|z > \\ < z|x > & < z|y > & < z|z > \end{array}$$

For an *sppp* shell quartet the ordering is

$< 1x|xx > \quad < 1x|xy > \quad < 1x|xz >$
$< 1x|yx > \quad < 1x|yy > \quad < 1x|yz >$
$< 1x|zx > \quad < 1x|zy > \quad < 1x|zz >$
$< 1y|xx > \quad < 1y|xy > \quad < 1y|xz >$
$< 1y|yx > \quad < 1y|yy > \quad < 1y|yz >$
$< 1y|zx > \quad < 1y|zy > \quad < 1y|zz >$
$< 1z|xx > \quad < 1z|xy > \quad < 1z|xz >$
$< 1z|yx > \quad < 1z|yy > \quad < 1z|yz >$
$< 1z|zx > \quad < 1z|zy > \quad < 1z|zz >$

Note that redundant integrals may be included. The ordering within a pure angular momentum buffer follows the same significance rule, with functions ordered in decreasing m_l ($l, l-1, ..., -l$).

For an *n*-center multiplet, a first derivative buffer contains a set of three derivative multiplets ($\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}$) for each of up to $n-1$ unique centers (at least one center omitted due to translational invariance).

For an *sssp* first derivative shell quartet (omitting derivatives with respect to center four) the ordering is

$\frac{\partial}{\partial x_1}(< 11|1x > \quad < 11|1y > \quad < 11|1z >)$
$\frac{\partial}{\partial y_1}(< 11|1x > \quad < 11|1y > \quad < 11|1z >)$
$\frac{\partial}{\partial z_1}(< 11|1x > \quad < 11|1y > \quad < 11|1z >)$
$\frac{\partial}{\partial x_2}(< 11|1x > \quad < 11|1y > \quad < 11|1z >)$
$\frac{\partial}{\partial y_2}(< 11|1x > \quad < 11|1y > \quad < 11|1z >)$
$\frac{\partial}{\partial z_2}(< 11|1x > \quad < 11|1y > \quad < 11|1z >)$
$\frac{\partial}{\partial x_3}(< 11|1x > \quad < 11|1y > \quad < 11|1z >)$
$\frac{\partial}{\partial y_3}(< 11|1x > \quad < 11|1y > \quad < 11|1z >)$
$\frac{\partial}{\partial z_3}(< 11|1x > \quad < 11|1y > \quad < 11|1z >)$

Similarly, for second derivatives the ordering is

$\frac{\partial^2}{\partial x_1^2}(< 11|1x > \quad < 11|1y > \quad < 11|1z >)$
$\frac{\partial^2}{\partial x_1 \partial y_1}(< 11|1x > \quad < 11|1y > \quad < 11|1z >)$
$\frac{\partial^2}{\partial x_1 \partial z_1}(< 11|1x > \quad < 11|1y > \quad < 11|1z >)$
$\frac{\partial^2}{\partial x_1 \partial x_2}(< 11|1x > \quad < 11|1y > \quad < 11|1z >)$
$\frac{\partial^2}{\partial x_1 \partial y_2}(< 11|1x > \quad < 11|1y > \quad < 11|1z >)$
$\frac{\partial^2}{\partial x_1 \partial z_2}(< 11|1x > \quad < 11|1y > \quad < 11|1z >)$
$\frac{\partial^2}{\partial x_1 \partial x_3}(< 11|1x > \quad < 11|1y > \quad < 11|1z >)$

$$\frac{\partial^2}{\partial x_1\partial y_3}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial x_1\partial z_3}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial y_1^2}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial y_1\partial z_1}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial y_1\partial x_2}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial y_1\partial y_2}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial y_1\partial z_2}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial y_1\partial x_3}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial y_1\partial y_3}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial y_1\partial z_3}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial z_1^2}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial z_1\partial x_2}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial z_1\partial y_2}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial z_1\partial z_2}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial z_1\partial x_3}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial z_1\partial y_3}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial z_1\partial z_3}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial x_2^2}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial x_2\partial y_2}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial x_2\partial z_2}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial x_2\partial x_3}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial x_2\partial y_3}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial x_2\partial z_3}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial y_2^2}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial y_2\partial z_2}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial y_2\partial x_3}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial y_2\partial y_3}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial y_2\partial z_3}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial z_2^2}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial z_2\partial x_3}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial z_2\partial y_3}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial z_2\partial z_3}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial x_3^2}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial x_3\partial y_3}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial x_3\partial z_3}(<11|1x>\quad<11|1y>\quad<11|1z>)$$
$$\frac{\partial^2}{\partial y_3^2}(<11|1x>\quad<11|1y>\quad<11|1z>)$$

$\frac{\partial^2}{\partial y_3 \partial z_3}(< 11|1x > \quad < 11|1y > \quad < 11|1z >)$
$\frac{\partial^2}{\partial z_3^2}(< 11|1x > \quad < 11|1y > \quad < 11|1z >)$

and likewise for higher order derivatives.

Normalization

Gaussian integral packages can have significantly different normalization conventions for the target integrals. The convention often depends on implementation details, such as the evaluation method. The normalization convention we adopted is “natural” for the majority of integral packages aimed at basis sets with segmented contractions.

Cartesian Gaussian functions in a shell of angular momentum L have the same normalization factor N. N is determined such that the Cartesian functions x^L, y^L, and z^L are normalized to unity. The norm of a Cartesian Gaussian $x^a y^b z^c$ is therefore

$$||x^a y^b z^c|| \quad = \quad \frac{(2a+2b+2c-1)!!}{(2a-1)!!(2b-1)!!(2c-1)!!}$$

All spherical harmonic Gaussians are normalized to unity. Transformation from Cartesian to spherical harmonics Gaussians was described in detail by Schlegel and Frisch.[30]