The component-based application for GAMESS

by

Fang Peng

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Ying Cai, Major Professor
Masha Sosonkina, Co-Major Professor
Mark Gordon
Ricky A. Kendall

Iowa State University

Ames, Iowa

2007

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1. INTRODUCTION

High performance scientific simulations in a wide range of areas, such as quantum chemistry, climate, high energy physics, earth observation and bioinformatics, often solve very complicated problems and require a large amount of resources. Most of the underlying programs for the scientific simulations have been under development for a long period of time; used different computing languages and programming models. As the new algorithms, methodology, and programming models in an area being created and upgraded, the corresponding scientific programs become more and more complicated. While each program is complicated by its own, the complexity can be hard to manage when several programs need to cooperate to perform the same task. The language interoperability also becomes an issue.

The Component Based Software Engineering (CBSE) aims to manage the complexity of a software system by using "plug-and-play" components. Those components are deployed based on software functionality and can interact with each other on component-based frameworks through the well-defined interfaces. The users are able to use the components without knowing which programming languages are used for implementing each component. The existing commercial available component-based frameworks include Microsoft's Component Object Model (COM) [1], the Object Management Group's Common Object Request Broker Architecture (CORBA) Component Model [2], and Sun's Enterprise JavaBeans [3]. However, none of those frameworks can handle high-performance architectures which are required for scientific programs.

Common Component Architecture (CCA) [4] was just designed for High Performance Computing (HPC). CCA offers an opportunity for scientific packages to dynamically interact with each other without manually dumping files, converting data formats or painstakingly coupling codes on a case-by-case basis. With CCA, scientists are able to construct new computations or improve the performance of their software by using components provided by other research groups through well-defined interfaces. This potential of interoperability encourages application scientists from different scientific domains to explore mechanisms to couple existing packages that offer different computing

IS-T 2710

ABSTRACT

GAMESS, a quantum chemistry program for electronic structure calculations, has been freely shared by high-performance application scientists for over twenty years. It provides a rich set of functionalities and can be run on a variety of parallel platforms through a distributed data interface. While a chemistry computation is sophisticated and hard to develop, the resource sharing among different chemistry packages will accelerate the development of new computations and encourage the cooperation of scientists from universities and laboratories. Common Component Architecture (CCA) offers an environment that allows scientific packages to dynamically interact with each other through components, which enable dynamic coupling of GAMESS with other chemistry packages, such as MPQC and NWChem. Conceptually, a computation can be constructed with "plug-and-play" components from scientific packages and require more than componentizing functions/subroutines of interest, especially for large-scale scientific packages with a long development history. In this research, we present our efforts to construct components for GAMESS that conform to the CCA specification. The goal is to enable the fine-grained interoperability between three quantum chemistry programs, GAMESS, MPQC and NWChem, via components. We focus on one of the three packages, GAMESS; delineate the structure of GAMESS computations, followed by our approaches to its component development. Then we use GAMESS as the driver to interoperate integral components from the other two packages, and show the solutions for interoperability problems along with preliminary results. To justify the versatility of the design, the Tuning and Analysis Utility (TAU) components have been coupled with GAMESS and its components, so that the performance of GAMESS and its components may be analyzed for a wide range of system parameters.

capabilities. Without such a component model, data exchange between two scientific packages can only be accomplished through a large amount of file recoding.

The standards of CCA are defined by the CCA Forum [5], a group of scientists from different national laboratories and academic institutes who are researchers in the high performance computing community. The CCA Forum aims to define the standards for the component-based frameworks for the high performance computing. It has developed several tastes of CCA frameworks, the supporting infrastructure and some general-purpose components. The language interoperability of CCA is enabled by Babel [6], a tool for solving the interoperability of components that are implemented in different programming languages such as FORTRAN, C, C++, Python, and Java. Babel relies on the Scientific Interface Definition Language (SIDL) for defining interfaces for scientific components.

Quantum chemistry is one of the scientific disciplines that are actively involved in exploring the interoperability capability offered by CCA. The complexity in quantum chemistry computations results in a large number of noncommercial packages developed by research laboratories and universities (The General Atomic and Molecular Electronic Structure System - GAMESS [7], MPQC [8], and NWChem [9] are three major quantum chemistry programs from DOE), each with unique capabilities and deficiencies. The development of a new method is usually very time-consuming thus it is an important task to integrate capabilities of different packages to develop new computations that are not possible with any single package.

While CCA offers an environment for scientific packages to interact with each other, a package must be componentized before it is able to provide/use components to/from other packages. With the long development history of quantum chemistry programs, efforts to their componentizing cannot be accomplished by any single research group. Scientists must join together to define a set of standardized interfaces and data structures for computations of interest, and then packages are to be componentized accordingly.

Even with the standardized interfaces and techniques provided by CCA forum, componentizing a package with a long development history itself poses a big challenge, which must be conquered before enabling interoperability between packages. While componentizing quantum chemistry programs on coarse-grain level was conducted in

previous studies [10], another important and useful approach for the quantum chemistry community is to componentized low-level computations such as molecular integral evaluations.

Molecular integral evaluation is a fundamental problem of all traditional quantum chemistry computations. The integral facilities available within one individual quantum chemistry program may lack one or more features of the others, limiting the range of methods which can be implemented and made 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 among various packages is the ability to implement new theoretical methods very rapidly.

In this thesis, I will give the background knowledge of this research in Chapter 2, including the basic concepts of quantum chemistry, the CCA terms, the parallel method used in GAMESS, and some special features of GAMES. In Chapter 3, several important CCA interfaces will be introduced and the corresponding components for each chemistry package, especially for GAMESS, will be explained in details. We developed the GAMESS CCA interface in two different parallel models: GAMESS/DDI and GAMESS/DDI/MPI models. GAMESS uses the Data Distributed Interface (DDI) [11] as its parallel communication mechanism, which mainly relies on TCP/IP sockets for communication. Integrating the GAMESS/DDI system with CCA is our first attempt to integrate GAMESS with the CCA framework. Besides TCP/IP sockets, the Message Passing Interface (MPI) [12] can also be used for DDI to enable GAMESS communications and a different mechanism has been developed for integrating GAMESS with MPI. In this mechanism DDI depends on MPI, instead of TCP/IP sockets, as the communication method. Since MPI is a widely used message passing interface, the GAMESS CCA components in this model are easily compatible with other components within CCA frameworks.

The componentizing mechanisms for several GAMESS computations: energy, gradient, Hessian, and integral computations, will be presented. The energy, gradient and Hessian computations have been incorporated into the *GAMESS.ModelFactory* component and the integral computation has been implemented in *GAMESS.IntegralEvaluatorFactory*

component. The strategies for wrapping the existing GAMESS code and implementation details of those GAMESS CCA components will be demonstrated.

Chapter 4 will cover the integration process of GAMESS with other scientific packages, including MPQC and NWChem, in the integral calculation. The discussion of the difficulties we encountered and preliminary experiment results will be presented in Chapter 5. In Chapter 6, we will conclude the research we have done and give the future works.

1.1 Common Component Architecture

The purpose of Common Component Architecture is to facilitate and promote the development of high performance scientific simulations with little programming requirements [4]. The CCA standard specifies just a minimal set of services that is required to be CCA compliant [5]. This design philosophy ensures the scientists focus on the implementation of components for a program instead of worry much about the interaction of components from different packages.

In the Common Component Architecture, the *components* are basic units of software that are composed together to provide a run-time component environment [5]. Instances of components are created and managed within a *framework*, which provides the basic services for components to operate and communicate with each other [5]. *Ports* are the fully abstract interfaces, through which components interact with each other and with the encapsulating framework [5]. A component must declare its *Provides* port to provide its own functions or services for other components to use, and also registers its *Uses* ports to connect references to *Provides* ports that are provided by other components or by the containing framework [5]. The communications between different components or between components and frameworks are enabled by connecting matched *Provides-Uses* port pairs through the framework.

Based on the requirements and restrictions from a wide range of scientific researches, several frameworks that compliant to CCA standards have been developed, each has unique features. There are two major types of CCA frameworks: *direct-connect* and *distributed* frameworks [5], where *direct-connect* frameworks do not have ability to manage components distributed on a wide area network, and *distributed* frameworks supports distributed components [5]. CCAFFEINE [14], developed by Sandia National Laboratory, is one of the

most commonly used CCA frameworks. It is a light-weight direct-connect framework that supports SPMD (Single Program Multiple Data) parallel computing model. Since CCAFFEINE was first developed, the CCA forum has continually upgrade it and provided tutorials and technical helps for helping scientists in a variety of area to create scientific components, it is the best choice for us to start the component development for quantum chemistry programs. Other CCA frameworks, such as DCA [15], DECAFE [16], CCAIN, which are direct-connect frameworks, and XCAT-JAVA [17], XCAT-C++ [18] and SCIRUN-2 [19], which are distributed frameworks, are also popular in some other research areas. I will only focus on the design of CCAFFEINE as it is the only one has been used for this research. In the future, we may extend chemistry components to be able to run on other CCA frameworks.

CCAFFEINE uses the peer component model, in which each component is treated independently without in a hierarchal relationship with other components. Components attach to a framework and connect with other components through *Provides-Uses* port pairs, which make them easier to be added or unplugged to/from a framework. When a CCAFFEINE framework is running in a parallel environment, each process has its own instance of a CCA framework, and an identical set of component instances and connections are loaded into each framework [4]. The set of similar component instances that are distributed across parallel processes can communicate with each other by using any available communication system (i.e. MPI, PVM [20], Global Arrays [21], or shared memory), while each framework instance that contains the identical set of component instances and connections manages the interactions among component instances within its own process [4]. Different sets of component instances are allowed to use different communication systems simultaneously under the same framework [4]; this is useful for the integration of legacy codes under CCA frameworks since legacy software usually has its own communication mechanisms.

1.2 Quantum Chemistry

Quantum chemistry is a subfield of theoretical chemistry that uses both physics and mathematical methods to solve the electronic structure of the molecule [22]. Molecules are

composed of positive charged nuclei and negative charged electrons. Different combinations of nuclei and number of electrons or different geometrical arrangements of nuclei in space form different kinds of molecules. Several primary problems that the quantum chemistry need to solve are: the geometrical arrangements of the nuclei that correspond to stable molecules; their relative energies and properties; the rate by which one stable molecule can transform into another; and the time dependence of molecular structures and properties [23]. However, the only systems that could be calculated correctly by using the quantum chemistry theory are those with one or two electrons, such as H_2^+ molecule. Therefore, different approximations are used for finding approximate solutions for different purposes.

There are two kinds of approximation methods in quantum chemistry: *ab initio* and *semi-empirical*. If solutions are generated without reference to experimental data, the methods are usually called *ab initio* (Latin: "from the beginning") [23]. The *ab initio* method is usually used for solving smaller molecules, since the calculations are very complex and time consuming, scaling formally as the fourth power of the size of the molecules. The *semi-empirical* method avoids some time consuming calculations, but uses some parameters generated from experimental measurements or by performing *ab initio* calculations [23]. GAMESS, MPQC and NWChem are three of the *ab initio* quantum chemistry programs.

The advances in both computer hardware and software have enabled some of theoretical methods to be translated into computer programs in order to produce real data that cannot otherwise be calculated by human hands. With computer programs, chemists do not have to remember every theoretical formula or understand every complicated calculation. They just enter the molecular geometry, the type of calculations, and some other features of a molecule, and wait for the results computed by computer programs. However, even for the same theoretical method, with different algorithms, hardware or computing models, different results may be produced. This variety of computations requires the users to choose the right set of parameters and methods to be able to get valuable results for a problem. Chemists often use the computing results to evaluate a large pool of experimental results or predict certain properties a molecule [23], instead of using it as the exact answers. There are many possible molecules and associate properties, but only a little portion of them have been evaluated by calculation or experiment. With the development of theoretical methods, better

algorithms, and the increasing computer power, the chemistry calculations can apply to more problems and become more accurate.

Because of the complexity of quantum chemistry calculations, many programs have been created by national laboratories and universities, while each program contains special capabilities. It is very complicated and time-consuming to create a new computation from scratch in a chemistry program, which may be already implemented in another program. The best computations provided in each package can be accessed by utilizing the interoperability capability provided by CCA through CCA components.

CHAPTER 2. BACKGROUND

GAMESS, NWChem and MPQC are three fundamental chemistry packages that are developed under the Department of Energy (DOE). The General Atomic and Molecular Electronic Structure System (GAMESS) is an *ab initio* quantum chemistry program, which was originally formed from HONDO5 and other programs at the Department of Energy's National Resource for Computations in Chemistry in the late 1970's [7].

Most of the source code of GAMESS is designed with FORTRAN 77. While portability can be achieved through this design (every modern cluster has a FORTRAN 77 compiler), incorporating an external module or interacting with other scientific packages can be very difficult since scientific packages developed in recent years seldom use FORTRAN 77 exclusively.

The Massively Parallel Quantum Chemistry Program (MPQC), written in the C++ programming language, computes properties of atoms and molecules from first principles. MPQC has been designed as a massively parallel program from the beginning, and it can run on a wide range of platforms, from UNIX workstations, symmetric multi-processors, to massively parallel architectures.

The class libraries underlying the MPQC program are written in C++ using an object-oriented design. Following a class hierarchy very similar to the CCA integral interfaces [24], the integral packages are encapsulated by integral evaluator and integral factory interfaces described within the MPQC documentation [25]. This encapsulation insures a clean separation of the integrals code which greatly simplified packaging the integral packages within MPQC as stand-alone components.

NWChem is a quantum chemistry program that is written in FORTRAN 77. It uses an object-oriented design and programming approach to facilitate functionality reuse and hide internal data. One example of this is the integral abstract programming interface (API) of NWChem. The API exposes only specific aspects of the integral computation to the programmer and hides many of the details with regard to which integral programs are used (there are currently four different algorithms within NWChem) and how the computations are done. This API has initialization routines that require the geometry and the basis set as well

as a termination routine that cleans up and terminates the integral computations. There is a set of routines based on the type of integrals to be computed (energy, first or second derivative). In addition, the API allows the programmer to select the accuracy (or the threshold for radial cutoffs) for the integrals. Once the API has been initialized there are specific routines to tell the programmer how much memory is needed for the buffers required by the API and then to call each of the different types of integrals that are available. This architecture allows any improvements or new integral routines to be automatically realized throughout the whole of NWChem.

NWChem also has basis set objects and geometry objects that must be properly populated so that the integral computations work. The population of these objects is usually initiated through an input file although they can also be created through functions associated with the objects. This is particularly useful in the context of CCA.

Each program has very different functionalities while sharing some common calculations. Instead of recoding a method from one program to make it work in another program, CCA provides a method to allow each program to access the functionalities of the other programs through pre-defined interfaces. In this research, we will focus on one of the chemistry programs: GAMESS, to detail the structure of the GAMESS computations, the communication model, and the procedure of componentizing GAMESS. As the base of understanding our work, several primary terms and calculations in quantum chemistry will be introduced in this section, followed by the structure of GAMESS computations and the parallel mechanisms of the Data Distributed Interface (DDI).

2.1 Quantum Chemistry Calculations

The heart of quantum chemistry theories is the time-independent **Schrödinger** equation, which in short-hand operator form [23] is given as

$$H\Psi = E\Psi \tag{2.1}$$

Where H is a Hamiltonian operator for a system of nuclei and electrons and it is independent of the time; E is the total energy; Ψ is the wave function that display both wave and particle characteristics of electronics. The square of the wave function gives the probability of finding the electron at a giving position [23].

The time-independent Schrödinger equation is used to solve the wave function for electrons and nuclei in space and their energies under certain circumstances. For every time-independent Hamiltonian operator, H, there exists a set of quantum states, Ψ_n, known as energy eigenstates, and corresponding real numbers E_n satisfying the eigenvalue equation [22],

$$H \mid \Psi_n \rangle = E_n \mid \Psi_n \rangle \tag{2.2}$$

The real number E_n is the eigenvalue of the Hamiltonian, also the total energy. The Hamiltonian operator contains operators for kinetic (T) and potential (V) energy of the nuclei and electrons.

$$H_{tot} = T_n + T_e + V_{ne} + V_{ee} + V_{nn} \tag{2.3}$$

$$T_n = \sum_a \frac{1}{2M_a} \nabla_a^2 \tag{2.4}$$

$$T_e = -\sum_1^N \frac{1}{2} \nabla_i^2 \tag{2.5}$$

$$V_{ne} = -\sum_i^N \sum_a \frac{Z_a}{|R_a - r_i|} \tag{2.6}$$

$$V_{ee} = \sum_i^N \sum_{j>i}^N \frac{1}{|r_i - r_j|} \tag{2.7}$$

$$V_{nn} = \sum_a \sum_{b>a} \frac{Z_a Z_b}{|R_a - R_b|} \tag{2.8}$$

R_a is the position vector for nuclei a. r_i is the position vector for electron i. Z_a is the atomic number of nuclei a. The Laplacian operator ∇_i^2 and ∇_a^2 involve differentiation with respect to the coordinates of electron i and nuclei a [22]. T_n is the operator for the kinetic energy of nuclei, T_e is the operator for the kinetic energy of electrons, V_{ne} is the operator for the coulomb attraction between nuclei and electrons, V_{ee} is the operator for the repulsion between electrons, and V_{nn} is the operator for the repulsion between nuclei.

As nuclei are much heavier than electrons and they move very slowly compare to electrons do, it is a good approximation to consider electrons moving in the field of fixed nuclei [23]. The Schrödinger equation is then separated into two parts: one part describes the

electronic wave function for a fixed nuclear geometry and another part describes the nuclear wave function [23]. This separation is called the **Born-Oppenheimer (BO)** approximation.

Within the Born-Oppenheimer approximation, the kinetic energy of the nuclei T_n can be neglected and the repulsion between the nuclei V_{nn} can be considered as a constant. Thus, the remaining terms are called the **electronic Hamiltonian**. The electronic Hamiltonian operator, H_e, for N electrons [23] is

$$H_e = T_e + V_{ne} + V_{ee} + V_{nn} \tag{2.9}$$

$$H_{tot} = T_n + H_e + H_{mp} \tag{2.10}$$

$$H_{mp} = -\frac{1}{2M_{tot}}\left(\sum_{i=1}^{N}\nabla_i\right)^2 \tag{2.11}$$

H_{mp} is called the mass-polarization, where M_{tot} is the total mass of all the nuclei and the sum is over all electrons. By the Born-Oppenheimer approximation, H_e depends only on the nuclear coordinates in space and not on their momentum. Thus, the electronic Schrödinger equation depends parametrically only on the nuclear coordinates [23].

The Born-Oppenheimer (BO) approximation introduces very small errors for most systems, while some effects have been implicitly neglected. Some correctness approaches can be performed after solving the electronic Schrödinger equation. The further details have been introduced in the classical quantum chemistry book: "Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory" that is written by Attila Szabo and Neil S. Ostlund [22].

2.1.1 Basic terms

The most common type of *ab initio* calculation is called a **Hartree-Fock** (HF) calculation, which is an approximate method for determining the ground-state wave function and ground-state energy of a quantum many-body system [22]. According to the variation principle, the approximate solutions for energies are always larger than or equal to the exact ground state energy, which means that the lower the energy, the better the wave functions [22]. The Hartree-Fock method aims to calculate the approximate energies by finding the approximate wave functions that minimizing the energies greater than or equal to the exact ground state energy. Considering the wave functions that depend on a set of parameters, we

can calculate the "best" wave functions by minimizing the energy that calculated by using a given set of parameters. The calculated energy equals to the exact ground state energy only if the given wave functions are the exact electronic spatial coordinates for the ground state [22].

The Hartree-Fock method

The most common type of *ab initio* calculation is called a **Hartree-Fock** (HF) calculation, which is an approximate method for determining the ground-state wave function and ground-state energy of a quantum many-body system [22]. According to the variation principle, the approximate solutions for energies are always larger than or equal to the exact ground state energy, which means that the lower the energy, the better the wave functions [22]. The Hartree-Fock method aims to calculate the approximate energies by finding the approximate wave functions that minimizing the energies greater than or equal to the exact ground state energy. Considering the wave functions that depend on a set of parameters, we can calculate the "best" wave functions by minimizing the energy that calculated by using a given set of parameters. The calculated energy equals to the exact ground state energy only if the given wave functions are the exact electronic spatial coordinates for the ground state [22].

The basis set approximation

In practices, the exact wave functions are impossible to get except for very small systems, such as one and two electron systems. Therefore, a set of known basis functions are normally used to express the unknown approximate wave functions. The basis function is a linear combination of primitive Gaussians, all of the same type and all on the same nucleus, but with different exponents:

$$\chi_\alpha = \sum_k d_{k\alpha} x^l y^m z^n e^{-\delta_k r^2} \tag{2.12}$$

Where k is the index of the primitive Gaussians, $d_{k\alpha}$ is a contraction coefficient, δ_k is the exponent, x, y, z are the Cartesian coordinates of the nucleus, and $r^2 = x^2 + y^2 + z^2$. The angular momentum of the shell type (S, P, D, F, G, ...) is given by $l + m + n$. For example, when $l + m + n = 0$, we get an S-type basis function,

$$\chi_\alpha = \sum_k d_{k\alpha} e^{-\delta_k r^2} \tag{2.13}$$

And, when $l + m + n = 1$, we have three types of different basis functions,

$$\chi_\alpha = \sum_k d_{k\alpha} x e^{-\delta_k r^2} \tag{2.14}$$

$$\chi_\alpha = \sum_k d_{k\alpha} y e^{-\delta_k r^2} \tag{2.15}$$

$$\chi_\alpha = \sum_k d_{k\alpha} z e^{-\delta_k r^2} \tag{2.16}$$

The formulas (2.14), (2.15) and (2.16) correspond to the P_x, P_y and P_z basis functions, respectively. Each set of basis functions are referred as an Atomic Orbital (AO). We define a Molecular Orbital (MO) as a linear combination of atomic orbitals. The MO may be written as [23]:

$$\phi_i = \sum_\alpha^M c_{\alpha i} \chi_\alpha \tag{2.17}$$

Where ϕ_i is a molecular orbital that forms from a linear combination of M atomic orbitals, χ_α; $c_{\alpha i}$ is a MO coefficient. The Hartree-Fock equations may be written as [23]:

$$F_i \sum_\alpha^M c_{\alpha i} \chi_\alpha = \varepsilon_i \sum_\alpha^M c_{\alpha i} \chi_\alpha \tag{2.18}$$

Where F_i is called the Fock operator, ε_i is the energy.

The Self-Consistent Field (SCF) techniques

The Hartree-Fock equations in the atomic orbital basis may be given in [23]:

$$F_{\alpha\beta} = \langle \chi_\alpha | F | \chi_\beta \rangle \tag{2.19}$$

The F matrix contains the Fock matrix elements. Each $F_{\alpha\beta}$ element is given as:

$$F_{\alpha\beta} = h_{\alpha\beta} + \sum_{\lambda\delta} G_{\alpha\beta\lambda\delta} D_{\lambda\delta} \tag{2.20}$$

Where $h_{\alpha\beta}$ denotes integrals involving the one-electron operators; $G_{\alpha\beta\lambda\delta}$ denotes the two-electron integrals involving the electron-electron repulsion operator; $D_{\lambda\delta}$ denotes the occupied MOs of coefficients, which is often referred as a density matrix [23]. The density

matrix can only be determined by diagonalizing the Fock matrix. On the other hand, the Fock matrix is only determined when all the occupied MOs coefficients are known. Therefore, the Fock matrix may be solved by starting from guessing a set of MOs coefficients and computing the Fock matrix literately.

Figure 1 shows how the Fock matrix is calculated by using its own solutions. First, the initial parameters (e.g. basis functions, molecular geometry, etc) are fed in and all one- and two-electron integrals are calculated. Then a suitable start guess for the MO coefficients are generated. The initial density matrix is calculated. The Fock matrix is formed from integrals and density matrix. By diagnosing the Fock matrix, the eigenvectors contain the new MO coefficients. This new MO coefficients will be fed into the system to form a new density matrix. If it is sufficiently close to the previous density matrix, we are done, otherwise we need to iteratively calculate the Fock matrix and generate new density matrix [23]. Thus, the Hartree-Fock method is also called the Self-Consistent Field (SCF) method.

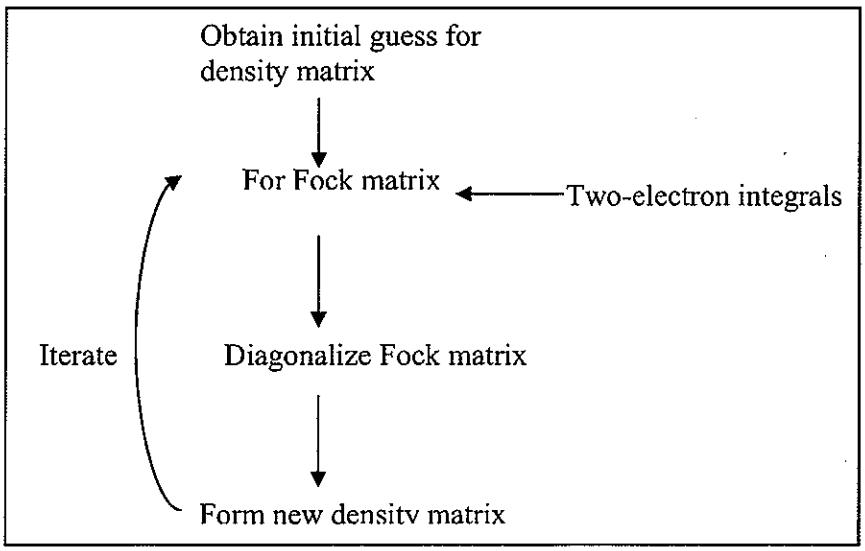

Figure 1. Illustration of the SCF procedure [23]

The Hartree-Fock method usually is considered as the starting point for more sophisticated methods. Either more approximations will be used, leading to a *Semi-empirical* method, or more basis functions are used to get a more accurate solution [23].

The evaluation of gradient and Hessian

The change in energy for moving a nucleus can be written as a Taylor expansion [23].

$$E(R) = E(R_0) + \frac{\partial E}{\partial R}(R - R_0) + \frac{1}{2}\frac{\partial^2 E}{\partial R^2}(R - R_0)^2 + \frac{1}{6}\frac{\partial^3 E}{\partial R^3}(R - R_0)^3 + \cdots \quad (2.21)$$

Where R is the nuclear geometry. The first derivative, $\frac{\partial E}{\partial R}$ is the **gradient** g, the second derivative, $\frac{\partial^2 E}{\partial R^2}$ is **the force constant (Hessian)** H etc [23]. A point is a stationary point if the gradient at that point is zero. If the *Ro* geometry is a stationary point, the force constant matrix may be used for evaluating harmonic vibrational frequencies and normal coordinates, q [23].

One- and two-electron integrals

The calculation of one-electron integrals (1- or 2-center integrals, where a center refers to a specific atom in a molecule) and two-electron integrals (1-, 2-, 3-, or 4-center integrals) is the basis of constructing the Fock matrix in any quantum chemistry program that uses the Self-Consistent Field (SCF) method.

Consider a molecule with N electrons. The nuclear-nuclear repulsion is a constant for a given nuclear geometry. The nuclear-electron attraction is the sum of terms, each depends only on one electron coordinate since the nuclei are fixed according to the Born-Oppenheimer (BO) approximation. The same holds for the electron kinetic energy. The electron-electron repulsion depends on two-electron coordinate. The operators may be collected according to the number of electron indices [23].

$$h_i = -\frac{1}{2}\nabla_i^2 - \sum_a \frac{Z_a}{|R_a - r_i|} \quad (2.21)$$

$$g_{ij} = \frac{1}{|r_i - r_j|} \quad (2.22)$$

$$H_e = \sum_{i=1}^{N} h_i + \sum_{i=1}^{N}\sum_{j>i}^{N} g_{ij} + V_{nn} \quad (2.23)$$

The one electron operator h_i describes the motion of electron i in the field of all nuclei, and g_{ij} is the two electron operator giving the electron-electron repulsion. The one-

and two-electron integrals in the atomic basis [23] are given in Eqs. (2.24) and (2.25), respectively:

$$\langle \chi_\alpha | h | \chi_\beta \rangle = \int \chi_\alpha(1) \left(-\frac{1}{2}\nabla^2 \right) \chi_\beta(1) dr_1 + \sum_a \int \chi_\alpha(1) \frac{Z_a}{|R_a - r_1|} \chi_\beta(1) dr_1 \tag{2.24}$$

$$\langle \chi_\alpha \chi_\gamma | g | \chi_\beta \chi_\delta \rangle = \int \chi_\alpha(1) \chi_\gamma(2) \frac{1}{|r_1 - r_2|} \chi_\beta(1) \chi_\delta(2) dr_1 dr_2 \tag{2.25}$$

Where χ is a basis function (or Atomic Orbital, or AO); α, β, γ, and δ are the indexes of the basis functions; h is the one-electron operator and g is the two-electron operator.

In practice, integrals are calculated in batches, where a batch is a collection of integrals having the same exponent (in this thesis, we use the term *Gaussian shell* or *shell* to represent a set of basis functions with the same exponent) [23]. For example, a <*pp*|*pp*> type batch has 81 individual integrals, where the basis function for a P-type shell has 3 types (3*3*3*3 = 81). We usually call a batch of one-electron integrals a *shell doublet* and a batch of two-electron integrals a *shell quartet.*

In short, to compute the one- and two-electron integrals, we need the **one-electron operator**, the **two-electron operator**, the **basis set** information, and the coordinates of the atoms in the molecule (**molecular geometry**). Different packages may use different techniques and can handle different sets of basis functions to calculate integrals.

2.1.2 Other important concepts

Use of symmetry. The *group theory* is a mathematical tool that often used in quantum chemistry for greatly simplifying applications by exploring the symmetrical properties in molecules [26]. The symmetric properties of a molecule can be identified by some symmetry operations that are performed on the molecule such that the position and orientation of the molecule before and after the operations are identical [26]. Those symmetry operations are grouped and labeled with specific symbols, including a proper axis of rotation (C*n*, *n* = 1,2,3, ...), the reflection through a plane (s), inversion through a center (i), the rotation about an axis followed by reflection through a plane perpendicular to that axis (S_n^k) [26]. For easily classifying the possible symmetrical operations associate with a molecule, the symmetry operations are grouped into different "*point group*". By entering a

point group, a quantum chemistry program can quickly decide to ignore some computations that will produce the same results due to the use of symmetry. For example, many one- and two-electron integrals for Fock operators can be ignored if the suitable linear combinations of basis functions have been formed (symmetry adapted functions) [23]. Almost all quantum chemistry programs use the symmetry to reduce the computation cost. Therefore, the use of symmetry is an important optimizing approach for a chemistry program and should be incorporated into the associate component implementation.

Integral screening. *Integral screening* is a technique to ignore calculating integrals that are estimated to have little or no contribution to the final results of the Fock matrix [23]. In practices, integral screening is normally done at the batch level, when the largest term of an integral batch is smaller than a given cut-off, the whole batch will be neglected [23]. Integral screening techniques are normally used as an optimizing mean in quantum chemistry programs, although the cut-off or thread hold for screening out integrals may be different in different programs.

Conventional & direct SCF. The number of two-electron integrals grows as the fourth power of the size of the basis set (the number of total basis functions, M). There are 8 different permutations for a two-electron integral $\langle x_1 x_2 | x_3 x_4 \rangle$ that are identical, so the total number of integrals can be less (approximately 1/8 of M^4) [23]. However, the disk space or memory that required for storing all the integrals will increase quickly while the size of the molecule increases. For example, a basis set with 100 basis functions generates ~ 12.5×10^6 integrals (each is a double precision floating point number), requiring ~ 100 Mbytes of disk space or memory [23]. When the number of basis functions grows to 200, there will be $\sim 25 \times 10^6$ integrals, and the required disk space or memory grows to ~ 1.5 Gbytes. When the size of a molecule is relative small, it may be possible for all the integrals to be stored in memory. This kind of approach is very efficient for performing a Hartree-Fock calculation. However, for larger molecules, the disk space was the only choice. In a *conventional* method, all of the integrals will be computed at once and stored in the disk for later calculations. In a *direct* method, the integrals will be computed and used immediately at each SCF iterate without storing to or reading from the disk. Traditionally, the conventional method was used for large molecules when a large amount of disk space was required and the performance of

CPUs was relatively slow. As the performance of CPUs increases quickly relative to the speed of the disk I/O, it is quite normal for direct SCF jobs to be faster than conventional SCF jobs.

2.2 GAMESS

GAMESS is able to solve a wide range of quantum chemistry computations including Hartree-Fock (HF) wave functions (RHF, ROHF, UHF), GVB, and MCSCF using the self-consistent field method [7]. It is installed on many high performance computing systems, including those at most DOE, DOD, and NSF supercomputer centers, many academic institutions, and widely in the private sector. It is also part of the standard benchmark suites employed, for example, by NERSC, by the High Performance Computer Modernization Program, and by several computer companies (e.g., IBM). By 2005, GAMESS had grown to roughly 650,000 lines of FORTRAN [27] and the number of GAMESS users is estimated to be on the order of 100,000.

Back in 1970's when GAMESS was developed; the top-down structured programming model was the primary software engineering methodology. In a top-down program, a large problem is broken into several sub-problems with each subprogram act independently to solve a sub-problem. Each subprogram in turn can be broken into smaller programs, and eventually, the flow of control reaches down to problems that can be solved directly, without further discompose. This programming model is simple and easy to use. However, the lack of data structures and the object-oriented design makes the code hard to be reused.

With such a top-down structure, componentizing GAMESS is not as easy as componentizing an object-orient program. We have to reorganize the structure of several GAMESS computations and comply with its parallel mechanisms to be able to integrate GAMESS and CCA frameworks. Since we cannot modify the original GAMESS codes, one strategy we used is to create an extra layer of codes – wrapper functions, to rewrite some GAMESS computations based on the original GAMESS codes. The methods from CCA interfaces invoke the wrapper functions, in stead of using GAMESS subroutines directly. The details about the wrapper functions and the CCA interfaces for GAMESS will be introduced

in the next chapter. In this section, some basic knowledge about GAMESS computations will be presented, including the structures of GAMESS computations, the memory allocation strategies, and the communication mechanisms for the Distributed Data Interfaces (DDI).

2.2.1 GAMESS Structures

A GAMESS computation starts by reading user input options from an external input file. GAMESS groups related input options into many namelist groups, and users have to follow the specified format and use pre-defined key words to customize the input information. The detailed input description can be found in the documents along with the GAMESS distributions.

Among the user input options, the type of wave functions (the theory), the basis sets and the molecular geometry are three kinds of the basic information that are required for all computations. In our experiments, we used the SCF theory for all of the computations since it is the starting point for more complicated or more accurate calculations. GAMESS can read basis sets from three different sources: from basis sets that are normally stored in GAMESS source code specified by the $BASIS group, from the $DATA group (both $BASIS and $DATA are groups of user input options), or from an external file. If the $BASIS group is omitted, the basis set must be given in the $DATA group input. The $DATA group describes the global molecular data such as point group symmetry, nuclear coordinates and possibly the basis set.

The memory allocation

When GAMESS starts, it allocates a large pool of memory from the system; the amount of memory can be decided by users from an input file or by the default value. If the memory is initialized correctly, a function can requests the amount of memory that is less than the available memory, and GAMESS will dynamically allocate the required amount of memory from the memory pool to the requester. This memory will be returned to the memory pool after being used and released. Figure 2 shows an example of this dynamical allocation of memory. The blue rectangle is the large memory pool allocated for GAMESS initially, which includes the part from the memory location *a* to the memory location *z*. When *subroutine1* needs to create an array of dimension *size1*, it will submit a request, *request1*, for

allocating a memory of size *size1*, where $b = a + size1$. If *size1* is less than the available memory, the memory from the location a to the location b will be reserved for *array1*. Again, if *subroutine1* sends another request for allocating memory of *size2* for *array2*, where $c = b + size2$ and c is less than z, the memory from the location b to the location c will be reserved for *array2*. The memory from the location c to the location z is still available. The requests for returning the memory of array1 and array2 have to be called later to avoid the memory leaking error.

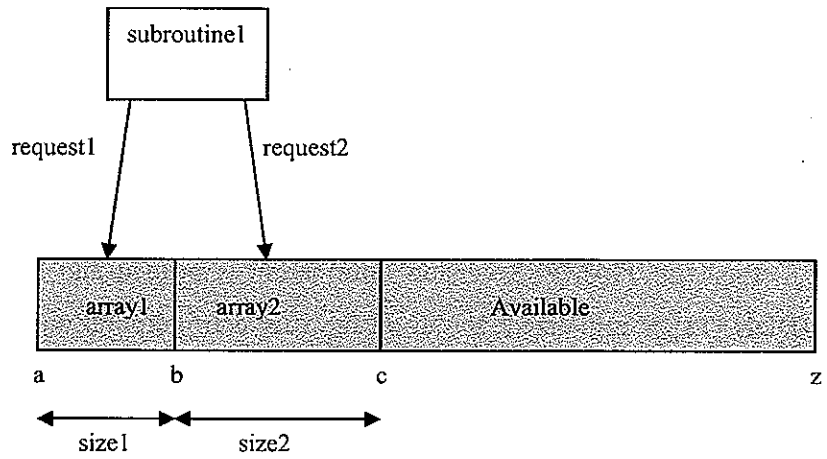

Figure 2. Memory allocation in GAMESS

Pass arrays between subroutines

Since there are no pointers or references used in FORTRAN 77, GAMESS passes the start location of an array in the memory pool and the size of the array to another subroutine as a parameter with the type of integer. The passed memory location in the other subroutine will be declared as an array instead of an integer. For example, when a subroutine, *subroutine2*, needs to use *array1* and *array2* (Figure 2) that allocated in *subroutine1*, the following two steps will be needed:

a. in *subroutine1*, call *subroutine2* by

```
CALL SUBROUTINE2(a, b, size1, size2)
```

b. in *subroutine2*,

```
SUBROUTINE2(a, b, size1, size2)
dimension a(size1), b(size2)
```

In this way, *a* and *b* can be used in *subroutine2* as arrays. The similar strategy for passing arrays between subroutines is also used in constructing GAMESS wrapper functions.

The sequence of a GAMESS computation

A GAMESS computation starts from the main subroutine and goes to a pre-defined branch based on the type of the computation. The global information, such as the program configuration, the basis set information and molecule coordinates, is stored as common blocks to be shared between subroutines. For some computations, intermediate data are stored as disk files to be used iteratively. The approach that GAMESS uses to handle global information complicates the componentizing process since we cannot simply pass pointers to global information between subroutines as in other object-oriented or modularized programs.

The execution sequence of the GAMESS main subroutine is shown in the left column of Figure 3. First, the GAMESS version information is printed and the Distributed Data Interface (DDI) [11] is initialized. Based on the user configuration during the compilation step, DDI choose to use TCP/IP sockets, MPI, or other communication libraries for communication.

Next, the calculation type, molecule coordinates, basis sets and other user input options are read from an external input file and the corresponding common blocks are initialized based on those inputs. Depending on the type of computation, the execution follows different branches, such as energy, gradient, Hessian, optimize, or saddle point. These computation branches are not independent from each other; one computation branch may overlap another branch. For example, a gradient computation needs to compute the energy first, so the route for the gradient branch will first go through the energy branch and then calculate the gradient. At the end of a computation, the control returns to the main subroutine for finalizing computations, cleaning up memory and finalizing the communication layer.

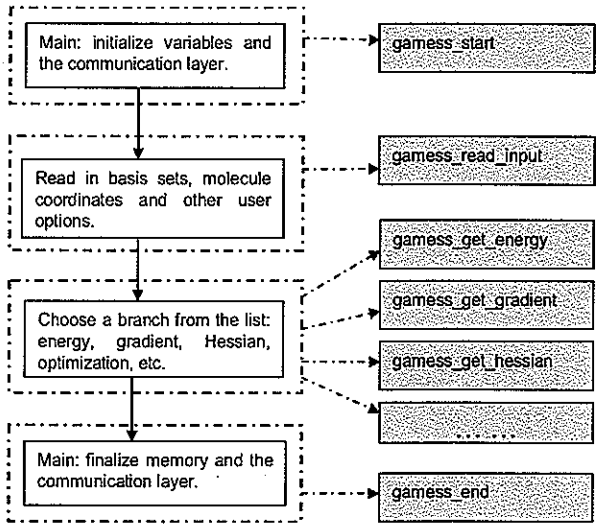

Figure 3. The execution sequence of the GAMESS main subroutine has four parts, shown on the left-side. Several wrapper functions (the right-side) are created by dividing the sequential main subroutine into smaller functions.

2.2.2 DDI

In the DDI communication model, two processes are normally assigned to a CPU, with one process performing the computational tasks, while the other exists solely to store and serve requests for the data associated with the distributed array [11]. There are some cases, in which a data server is not required, such as when using DDI over one-sided message libraries[1]. Also, with the latest version of DDI, the data server is not required when MPI/MPI2 or ARMCI is used as a communication mean[2]. **In Section 2.2.2, I only consider the cases when the data server is needed**, since the design of compute process/data server is a special feature in DDI and is hard to understand.

On a SMP machine or cluster (Figure 4), all the DDI processes (both compute and data server processes) within a node have direct access to all distributed array segments in the shared memory of that node. Thus, each compute process and data server can use system shared memory operations, such as copy or paste, locally to access the portion of a distributed array in its local shared memory without using any parallel communication

[1] DDI relies on LAPI or SHMEM libraries rather than TCP/IP on some high-end parallel systems
[2] For this version of DDI, only the ARMCI model has been used in the official distribution of the GAMESS program

mechanisms. Depending on the platform, communications between compute processes and data servers among different nodes occur either via TCP/IP sockets connections or MPI/MPI-2 [12]. When DDI uses TCP/IP sockets for communication, the DDI kickoff program is used for starting the required number of processes on every requested machine in the cluster that will run the job. If MPI/MPI-2 is used as the communication mechanism, then *mpirun* (or *mpiexec*) is used to start GAMESS processes.

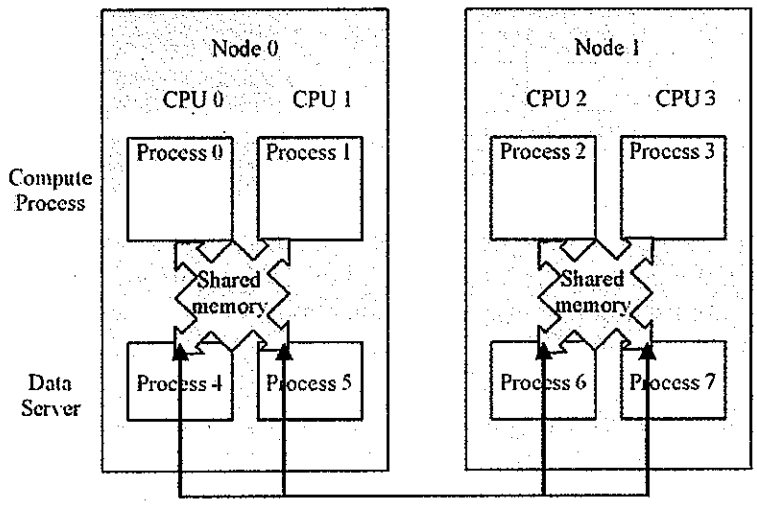

Figure 4. When DDI is used on an SMP cluster, all DDI processes within a node can access the distributed array in the node. The communications between data servers among different nodes depend on the communication mechanism configured with DDI (i.e., TCP/IP sockets, or MPI/MPI-2) [11].

Figure 5 shows the sequence of how the DDI kickoff program starts GAMESS or other programs. First, the DDI kickoff program needs the program name and the host list as command-line arguments; the host list is a list of host machine names and the number of CPUs in each node. The master DDI kickoff process analyzes the host list to catch the information on how many compute processes and data servers reside on each host machine. Second, a copy of the DDI kickoff program, along with information about host machines is spawned on each remote host in binomial order. As soon as a copy of the DDI kickoff program is launched on a host node, it creates the requested number of compute and data server processes on that host machine. Finally, a copy of the GAMESS program, with the host machine list, socket ports, host machine and process identities as the command-line

arguments, starts on each computer and data server process. The TCP/IP socket connections between a DDI kickoff process and a compute or data server process on the same host machine is created after the program starts the DDI initialization procedures. The DDI kickoff process on each host machine will wait for each compute and data server process to check in by listening to TCP/IP socket connections. As soon as all compute and data server processes are checked in, the communication is established for all compute and data server processes.

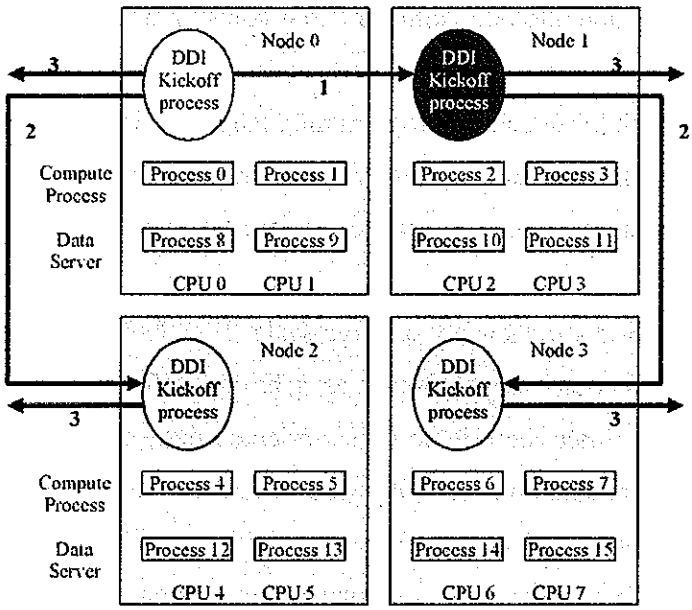

Figure 5. The numbers along with the arrows show the sequence of how the DDI kickoff program starts the remote DDI kickoff processes. First, the DDI kickoff program starts the master DDI kickoff process (the white one) in Node 0. Then, it starts a copy of remote DDI kickoff process (the blue one) in Node 1. Both DDI kickoff processes in Node 0 and Node 1 will send commands to start the remote DDI kickoff processes (the yellow ones) on Node 2 and Node 3. Next, all the DDI kickoff processes will start the remote DDI kickoff processes in other Nodes if needed. The same procedure will continue until all the required nodes have a copy of DDI kickoff program running. Finally, each copy of the DDI kickoff program will create one compute process and one data server process on each CPU and GAMESS (or other programs) will be running in each compute/data server process.

CHAPTER 3. COMPONENTS IMPLEMENTATION FOR GAMESS

In general, the first step of componentizing a package is to create the SIDL interfaces. In our case, we need to extend the pre-defined chemistry interfaces in the *cca-chem-generic* package [28]. Next, the implementation files of the specified programming languages (C, C++, f77, f90, python, or java) are generated based on those interfaces by using Babel, the language interoperability tool. The auto-generated implementation files initially contain no customized codes; they include only some splicing banners and some auto-generated codes and comments. Programmers need to insert codes between splicing banners in each implementation file with the specified programming language; in our case C++. During each compilation, those implementation files will be regenerated according to the SIDL definitions, but the customized codes that are inserted between splicing banners will not be modified.

In addition, to componentize a large-scale FORTAN 77 based code such as GAMESS, wrapper functions are necessary as a bridge between CCA interfaces and the native GAMESS code. Since there is no object-oriented design in the GAMESS code, it is difficult for the implementation of GAMESS CCA components to utilize GAMESS subroutines directly. The use of wrapper functions divides GAMESS subroutines into smaller and less interleaving functions and therefore makes the componentization possible.

However, simply implementing the chemistry interfaces is not enough for GAMESS to run under the CCA framework, since GAMESS relies on DDI to start the computation, either sequential or parallel. We first need to construct communication models that allow DDI to run under the CCA framework. When DDI relies on TCP/IP as communication methods, the DDI kickoff program is used to kick off the corresponding program, GAMESS in our cases. Thus, we constructed our first communication model for the GAMESS CCA components: the "GAMESS/DDI" model, where the DDI kickoff program will start the CCA framework in each process. The GAMESS/DDI model was useful when we started implementing the CCA interfaces for GAMESS, since it was the easiest and the most straightforward way to make the GAMESS CCA components work under the CCA framework. The limitation of the GAMESS/DDI model is that only the programs that use

DDI as the communication interfaces are able to run in parallel. The second communication model, therefore, is created, in which DDI depends on MPI/MPI-2 as the underlying communication layer. We call the second model the "GAMESS/DDI/MPI" model. In this model, the MPI startup program is used to kickoff the required processes and any programs that use MPI are able to run in parallel.

In this section, I will introduce several commonly used CCA chemistry interfaces that defined in the *cca-chem-generic* package; followed by the detailed description and analysis of the two communication models for GAMESS CCA components; finally, the implementation procedure of several GAMESS CCA components will be demonstrated in details.

3.1 CCA Chemistry Interfaces

Most quantum chemistry programs perform fundamental chemistry calculations. Although existing chemistry packages may have a lot of overlapping functionalities, some of them may be more efficient in certain calculations while others may provide special functionality. The CCA provides an environment for different quantum chemistry programs to communicate with each other, and opens the possibility to utilize the best of each package. The CCA Chemistry group [28] already integrates several quantum chemistry programs, optimization solver packages, and parallel data management packages to perform geometry optimizations.

A set of chemistry interfaces are defined in the ***cca-chem-generic*** package [28] that each chemistry package can implement to create chemistry components and classes. In the design of those chemistry interfaces, the interface for a "component" usually ends with "*FactoryInterface*" and the corresponding component usually acts as a driver to return references to some classes, while a "class" usually provides real computations. The implementation of a component is only different from the implementation of a class in that a component also needs to implement the *gov.cca.Component* and *gov.cca.Port* interfaces.

ModelInterface & ModelFactoryInterface. The *ModelInterface* declares the primary functions in quantum chemistry computations, such as the evaluation of molecule energies, gradient and Cartesian Hessians. The *ModelFactoryInterface* declares methods to

provide model options and initializes the *model* class. Basically, a *ModelFactory* component (implements *ModelFactoryInterface*) will be initialized with the user input options, such as the type of theory, the basis sets, etc. The *get_model* mothod could then be invoked to get a *model* class (implements *ModelInterface*). The *get_energy*, *get_gradient*, and *get_hessian* methods are three primary methods for a *model* class to perform those chemistry calculations.

MoleculeInterface & MoleculeFactoryInterface. The *MoleculeInterface* declares functions for gathering information of a molecule, such as Cartesian coordinates and atomic number. The *MoleculeFactoryInterface* declares functions to instantiate molecule classes [28]. The *cca-chem-generic* package provides the implementation for the *Chemistry.MoleculeFactory* component (implements *MoleculeFactoryInterface*) and the *Chemistry.Molecule* class (implements *MoleculeInterface*) for all packages to use.

Integral evaluation interfaces. There are four core interfaces for integral computations: *IntegralEvaluator1Interface* for 1-center integrals, *IntegralEvaluator2Interface* for 2-center integrals, *IntegralEvaluator3Interface* for 3-center integrals and *IntegralEvaluator4Interface* for 4-center integrals. We call any classes that implement the above interfaces integral evaluators. Another core interface is *IntegralEvaluatorFactoryInterface*, which serves as a driver that returns references to the integral evaluators. An integral evaluator factory that implements *IntegralEvaluatorFactoryInterface* usually also extends the *gov.cca.Component* and *gov.cca.Port* interfaces and is used to provide integral evaluators for each chemistry package. Figure 6 shows the relationship among those five core integral interfaces and the three chemistry packages.

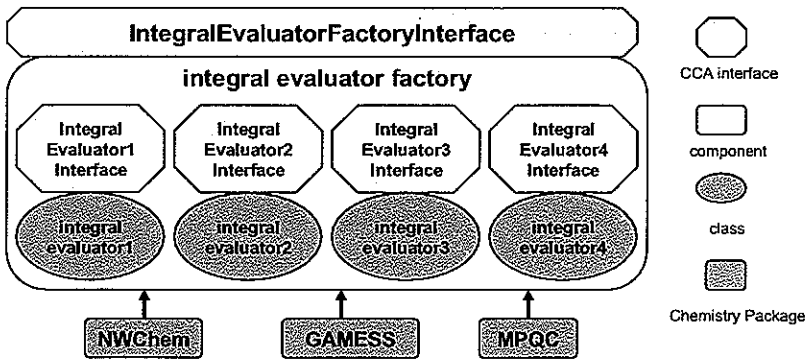

Figure 6. Each chemistry package can implement the *IntegralEvaluatorFactoryInterface* to provide an *integral evaluator factory* component and implement one or more of *IntegralEvaluatorNInterface* (*N*=1, 2, 3 and 4) to provide the *integral evaluatorN* classes. The *integral evaluator factory* component is a driver component to return the references to *integral evaluators* for integral computations.

An *integral evaluator* interface provides a *compute* method for calculating integrals for a shell multiplet. For example, the *compute* method of *IntegralEvaluator2Interface* is for computing a shell doublet, which is illustrated below,

```
/** Compute a shell doublet of integrals.
@param shellnum1 Gaussian shell number 1.
@param shellnum2 Gaussian shell number 2. */
void compute(in long shellnum1, in long shellnum2);
```

Where two indexes of Gaussian shells are passed as parameters and the resulting integrals are stored in a buffer that is initialized by the integral evaluator. Similarly, the *compute* method of *IntegralEvaluator4Interface* needs four indexes of Gaussian shells as parameters to compute integrals for a shell quartet.

Several auxiliary interfaces. Several auxiliary interfaces are also important to the initialization of integral evaluators: *CompositeIntegralDescrInterface*, *MolecularInterface*, *AtomicInterface* and *ShellInterface*. The *IntegralDescrInterface* is used to configure integral evaluators, which stores the information such as the type of integrals and derivative centers. The *MolecularInterface* provides a *molecule* (implements *MoleculeInterface*) object and the atomic basis data for a molecular Gaussian basis set, which includes the atomic basis set for any atom number of the molecule. The *AtomicInterface* provides the shell data for an atomic Gaussian basis set (AO), which provides a Gaussian shell for any given shell number. The *ShellInterface* provides the primitive and contraction data for a Gaussian shell [24]. Through

these interfaces, the information required for computing integrals can be passed from one package to another package without initializing every package. Figure 7 shows an example of how molecule coordinates and the basis set are stored in CCA integral objects.

ATOM	ATOMIC CHARGE	COORDINATES (BOHR) X	Y	Z
O	8.0	0.0000000000	0.0000000000	0.1239321808
H	1.0	1.4305200000	0.0000000000	-0.9834468192
H	1.0	-1.4305200000	0.0000000000	-0.9834468192

SHELL	TYPE	PRIMITIVE	EXPONENT	CONTRACTION COEFFICIENT(S)	
O					
1	S	1	130.7093214	0.154328967295	
1	S	2	23.8088661	0.535328142282	
1	S	3	6.4436083	0.444634542185	
2	L	4	5.0331513	-0.099967229187	0.155916274999
2	L	5	1.1695961	0.399512826089	0.607683718598
2	L	6	0.3803890	0.700115468880	0.391957393099
H					
3	S	7	3.4252509	0.154328967295	
3	S	8	0.6239137	0.535328142282	
3	S	9	0.1688554	0.444634542185	
H					
4	S	10	3.4252509	0.154328967295	
4	S	11	0.6239137	0.535328142282	
4	S	12	0.1688554	0.444634542185	

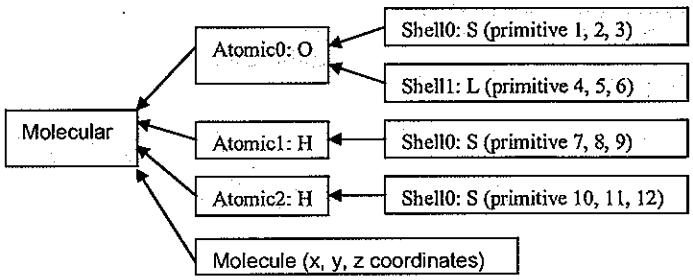

Figure 7. When using the water molecule and the "STO-3G" basis set as inputs, the information of molecule coordinates and the molecular basis sets in the GAMESS program is shown in the upper table. The upper block of the table shows the X, Y, Z coordinates of the water molecule. The bottom block of the table contains several columns. The information shown in the order from left to right is: the atomic symbols, the index of Gaussian shells, the Gaussian shell types, the primitive Gaussian shells, the exponents and contraction coefficients. Following each atom symbol is a block of Gaussian shells associated with it. The corresponding CCA integral components that store the same information are shown in the lower graph. The molecule coordinates are stored in a *Molecule* object (implements *MoleculeInterface*). The basis set information is stored in three *Atomic* (implements *AtomicInterface*) objects with the references to the corresponding *Shell* (implements *ShellInterface*) objects. A *Molecular* (implements *MolecularInterface*) object contains the references to the *Molecule* object and three *Atomic* objects.

An example of applications. Figure 8 shows an application example of the chemistry components under the CCA framework. The *MoleculeFactory* component, *ModelFactory* component and a *driver* component are instantiated under a single CCA framework. The *MoleculeFactory* component can get the reference of the *Molecule* class through the *Provides* port of the *MoleculeFactory* component and invoke the methods of the *Molecule* class. Similarly, the *driver* component can get the reference of the *Model* class that instantiated and initialized by the *MoleculeFactory* component, and then invokes the methods of the *Model* class, such as *get_energy*, *get_gradient*, and *get_hessian*. The *driver* component will also output calculation results returned from the *Model* class.

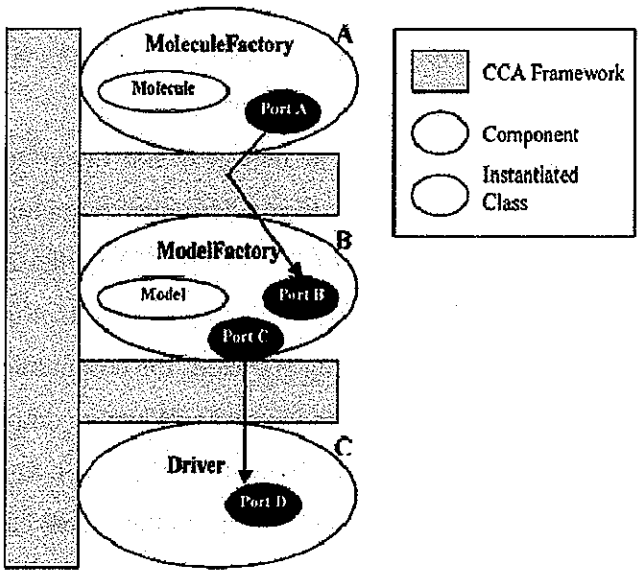

Figure 8. Port A is a *Provides* port that is implemented by the *MoleculeFactory* component, through which the reference of the *Molecule* class is passed to other components. Port C is a *Provides* port that implemented by the *ModelFactory* component, through which the reference of the *Model* class is passed. Port B and port D are *Uses* ports that are registered by the *ModelFactory* component and the *driver* component for using the services provided by other components.

3.2 Mechanisms of Creating GAMESS CCA Components

GAMESS requires DDI as the communication library when running in both sequential and parallel. DDI mainly relies on TCP/IP sockets for communication, and can also use MPI/MPI-2 as its underlying communication mechanisms. When a GAMESS CCA

component is instantiated under a CCA framework, it requires DDI being initialized to be able to use GAMESS functions. It is thus important to integrate DDI with the CCA framework to enable GAMESS CCA components running in both sequential and parallel.

Since TCP/IP is the most commonly used mechanism used DDI for GAMESS that installed on most of architectures, we start integrating DDI and the CCA framework by using the TCP/IP sockets as the underlying communication methods - the GAMESS/DDI model. The GAMESS/DDI model works fine for the components that use DDI as the communication library. However, for the components that do not use DDI, the GAMESS/DDI model restricts them for running in parallel. For example, since the DDI kickoff program is required for starting DDI processes in the GAMESS/DDI model, the MPI startup program cannot be used for starting processes and components that rely on MPI/MPI-2 for communications are not allowed to run in parallel.

The GAMESS/DDI/MPI model is designed for integrating DDI with the CCA framework when MPI/MPI-2 is used as the underlying communication library. However, when the data server is required, the GAMESS/DDI/MPI model raises problems for some components that depend on all MPI processes for computations. Since half of processes will be assigned as data servers, purely serving the calls for communication requests, there are only half of processes performing computation tasks. When a component needs to perform a global computation, such as a global sum calculation, they will wait for the results from the half of processes (data servers) that will never perform the global calculation, and a deadlock occurs.

This problem can be avoided if no data server is required, all of the allocated processes being used for computations. There is a version of DDI (newly developed) does not require the data server when relies on MPI/MPI-2 or ARMCI. We will introduce GAMESS/DDI/MPI model in both cases: the cases when the data server is required and when it is not required.

3.2.1 The GAMESS/DDI mechanism

When more than one CPU is required, the DDI kickoff program starts a compute process/data server pair for each CPU. An instance of the CCA framework is started on each compute process. The data servers are put to sleep and purely wait for the communication

requests from compute processes. Each instance of the CCA framework will initialize components and the connections among components based on user inputs. All the components and connections contained in a framework are identical on each compute process. The GAMESS CCA components contained in the framework of each compute process will initialize the DDI communication layer, in which only the components or the underlying programs that use DDI as the communication mechanism are able to run in parallel. The CCA framework or other components under the same framework cannot run in parallel, since the communication mechanisms used by the CCA framework or other components, such as MPI/MPI-2, will not be initialized.

Figure 9 shows a simple example of the GAMESS/DDI model under the CCA framework. On a SMP cluster with 4 nodes, the DDI kickoff process (section 2.2.2) on each node starts one compute process/data server pair for each CPU of that node, and then each compute process starts an instance of the CCA framework. The CCA framework, the component instances and their connections that are contained in the CCA framework are identical for all compute processes. When the DDI initialization procedure succeeds and the communication layer of DDI is established, the GAMESS CCA components within the same node can directly access the distributed arrays that are stored in the local shared memory of that node, and the GAMESS CCA components among different nodes can communicate with each other by using TCP/IP. The underlying communication operations are performed by the data server that associated with each compute process.

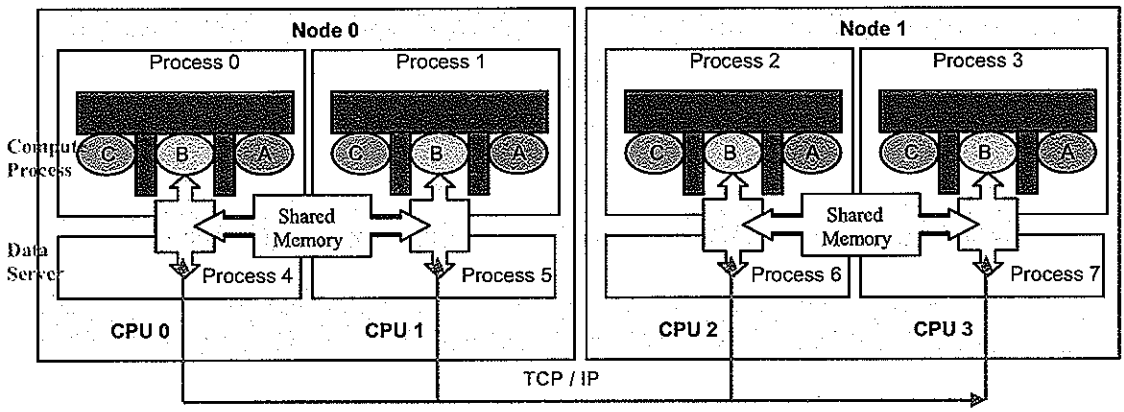

Figure 9. Under this model, one compute process/data server pair is created for each CPU. The CCA framework (green part) is initialized on each compute process. The data servers are put to sleep and purely waiting for the requests for the communication from compute processes. A is the driver component, which gets the *Model* object from B (the GAMESS component) through *Provides/Uses* ports. C is the *MoleculeFactory* component, which provides the molecule object to the GAMESS CCA component. The yellow area is the portion of distributed arrays that stored in the local shared memory of a node, where the compute processes and data server processes can directly access. The communication of compute processes among different nodes is through the TCP/IP sockets connections.

The major difficulty we encountered in designing this model is passing command-line arguments, which contain the information for initializing the DDI program (Section 2.2.2), from the DDI kickoff program to the GAMESS CCA components. When the DDI kickoff program starts the CCA framework, the command-line arguments are passed to the CCA framework, and there is no way to pass the arguments directly to a component under the CCA framework. Without the command-line arguments, DDI initialization cannot connect with the corresponding DDI kickoff program in that host machine, and the communication layers cannot be established correctly. Therefore, the "Stovepipe" Library provided by the CCA framework is used to convey the argument list from the CCA framework to the GAMESS CCA components.

Even though the GAMESS/DDI model works fine for GAMESS CCA components, it prohibits the components from other packages that do not use DDI from running in parallel. The GAMESS/DDI/MPI model is necessary for GAMESS to interact with other packages through the CCA framework.

3.2.2 The GAMESS/DDI/MPI mechanism

With the data server

The current version of DDI (the version is used with GAMESS) requires the data server when relies on the mix of MPI/TCP. In this model, the MPI startup program will initialize the required processes, where half of allocated processes are specified as "compute process" and half of processes are assigned as "data server". For example, when running the CCA framework with $2n$ processes, the DDI initializing procedure will use only n processes for computations and n processes for the communication. The processes within the same node can direct access to the portion of distributed arrays in the local shared memory of that node. This is different from the GAMESS/DDI model in two ways. First, both MPI and TCP/IP are used for communication. MPI is used to pass the actual data, such as a part of distributed arrays, when a process tries to access the portion of the distributed arrays that is not in its local shared memory. The TCP/IP is used for some smaller messages, such as a system call for waking up a sleeping process. The mixed message passing method is used, since most MPI implementations require a process to continuously check for the incoming calls. Thus, using pure MPI will make a data server compete for CPU resources with compute processes. In the TCP/IP implementation, while waiting for a request, each data server process is put to sleep, thus essentially yielding full CPU access to the compute process [11]. Therefore, the mixed MPI/TCP model for DDI should out-perform using pure MPI.

The GAMESS/DDI/MPI model for GAMESS CCA components is based on the MPI/TCP model for DDI. This model allows GAMESS and other programs to run in parallel through MPI/MPI-2 calls when running under the same CCA framework. However, with the requirement of the data server by DDI, the other programs may have trouble to run correctly in parallel since half of allocated processes have been put to sleep. For example, the MPQC program knows that the number of processes in MPI_COMM_WORLD is $2n$, while only n of processes are performing computations. When MPQC doing a parallel calculation, such as the global sum, it will by default use all $2n$ processes for the computation, but the number of actual processes that running MPQC components is only n; the other n processes are assigned as data servers and do no real computations. This will cause the deadlock in MPQC for

waiting for the results from *n* data servers that will never perform the task.

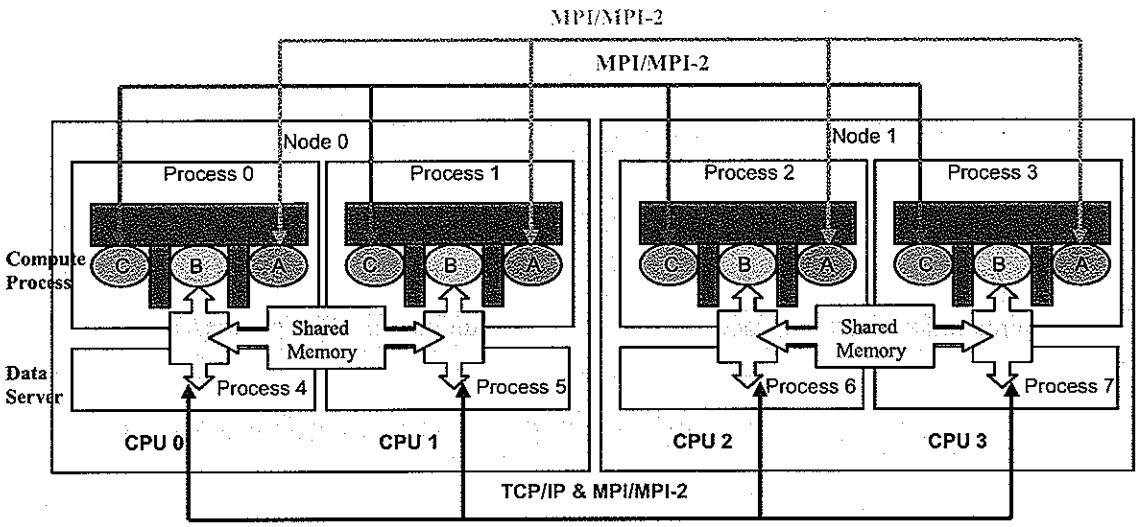

Figure 10. Under the GAMESS/DDI/MPI model, half of the processes is assigned as compute process and half of the processes is specified as data server. The CCA framework (green part) is initialized on each compute process. The data servers are put to sleep and purely waiting for the requests for the communication from compute processes. A is the driver component, which gets the *Model* object from B (the GAMESS component) through *Provides/Uses* ports. C is the *MoleculeFactory* component, which provides the molecule object to the GAMESS CCA component. The yellow area is the portion of distributed arrays that stored in the local shared memory of a node, where the compute processes and data server processes can directly access. The communication of compute processes among different nodes is through TCP/IP sockets or MPI/MPI-2. Components A and C are able to communicate among compute processes through MPI/MPI-2.

Without the data server

A newer version of DDI has eliminated the requirement of the data server when using MPI/MPI-2 or ARMCI as the underlying communication library. When this version of DDI relies on MPI/MPI-2, it purely uses MPI/MPI-2 calls for communication, not depending on TCP/IP. When using the older version of DDI with MPI/MPI-2, half of the allocated processes are assigned as data servers. If GAMESS works with other programs that use MPI/MPI-2, the programs other than GAMESS may enter the deadlock when they expect "data servers" should do the same computations as "compute processes" do, since data servers are used purely for communication, no computations are allowed. By eliminating the

data server when relying on MPI/MPI-2, DDI is able to work with other programs without the restriction caused by the data server. It thus allows GAMESS to cooperate with other programs by using MPI/MPI-2 through CCA components.

The upgraded GAMESS/DDI/MPI model for the GAMESS CCA components is based on the newer version of DDI when it depends on MPI/MPI-2. This model uses the MPI startup program to initialize the required processes. The sequences for initializing the CCA framework and GAMESS computations are similar with the GAMESS/DDI/MPI model with the data server, except that all of the processes are performing computations (no data server). The CCA framework and components that use MPI/MPI-2 as the communication method will be able to run in parallel by using this model without any restrictions from GAMESS/DDI.

3.3 GAMESS CCA Components

GAMESS has implemented several chemistry components, including *GAMESS.GaussianBasisMolecular*, *GAMESS.GaussianBasisAtomic*, *GAMESS.GaussianBasisShell*, *GAMESS.ModelFactory*, *GAMESS.Model*, *GAMESS.IntegralEvaluatorFactory*, *GAMESS.IntegralEvaluator2*, and *GAMESS.IntegralEvaluator4*. To be able to use GAMESS functions, the wrapper functions are required as bridges between GAMESS and the CCA interfaces. There are four groups of wrapper functions have been created according to their functionalities: (1) initializing the GAMESS program and DDI; (2) initializing the basis set information; (3) calculating energy, gradient and Hessian; (4) calculating 1e- and 2e-integrals. The implementation of those wrapper functions is different from cases to cases, depending on the implementation of the corresponding GAMESS subroutines and the SIDL interfaces for GAMESS CCA components.

The implementation of GAMESS CCA components is straightforward for most of methods, just invoking the corresponding wrapper functions. The wrapper functions can be considered as a part of the implementation for GAMESS CCA components. The implementation files in the server-side for GAMESS CCA components are initially empty, being auto-generated by BABEL based on SIDL interfaces. To insert codes into those

implementation files, the corresponding wrapper functions are invoked for performing specific calculations in GAMESS. For example, to implement *get_energy* method of *GAMESS.Model* class, the *gamess_get_energy* wrapper function is called inside the *get_energy* mothod. Thus, the wrapper functions can be considered as the part of implementation for GAMESS CCA components, or as the extra layer of function calls between the component implementation files and the original GAMESS subroutines.

In this section, we will present the procedure of constructing wrapper functions, the implementation of the GAMESS CCA components, and finally the structure of GAMESS CCA components.

3.3.1 The design of GAMESS wrapper functions

The layer of wrapper functions is between the CCA interfaces (the implementations for GAMESS CCA components) and the original GAMESS codes. The wrapper functions are created based on CCA SIDL interfaces and the underlying structure of GAMESS subroutines. When a method defined in a SIDL interface that require the specific information from the GAMESS program, such as the exponent for a primitive Gaussian, a corresponding wrapper function is created for reading the exponent from the associated common block in the GAMESS program. The method in CCA side (the implementation files) will invoke this wrapper function, instead of directly reading the common block from the GAMESS program. There are several reasons that we require wrapper functions.

First, the GAMESS program adopts a top-down programming model and there is no object-oriented or modularized design concepts built in. A computation (a branch) is usually started from a driver subroutine and continued with several sub-branches based on the user input option or the default settings. The codes for those sub-branches usually interleave with each other or depend on the results computed by branches. When a CCA method needs to access a sub-problem, instead of the whole computation, there are no subroutines that we can use directly to calculate the sub-problem without touching common blocks or codes in other subroutines. These tightly interleaving codes for GAMESS computations make the componentizing procedure a hard task. If we reorganize the part of codes for solving a sub-problem and group them into wrapper functions with the modularized design, it is possible

for us to invoke these wrapper functions from the CCA method. Otherwise, there is no way that we can componentize a computation in GAMESS. Moreover, we can test the wrapper functions for a GAMESS computation without touching the CCA implementations, and we only need to test if the invoke/return processes are correct.

Second, the wrapper functions can be accessed easily through the function headers for multiple times without touching the real codes. When a newer version of GAMESS is distributed, the corresponding codes in wrapper functions are also required to be upgraded. We need to manually modify/test wrapper functions or create an automatic tool to perform this task. The whole concept of CCA is for the reusability and interoperability between software systems. It would be easier to manage the code if we use a GAMESS computation through the wrapper functions, instead of inserting GAMESS codes directly into the CCA implementation files.

Finally, the current GAMESS CCA components are implemented in C++, and the GAMESS code is written in FORTRAN 77. The wrapper functions are necessary as the middle layer of the function calls in between the C++ component implementation and the FORTRAN 77 GAMESS program. The following details several strategies we used for constructing wrapper functions.

Initializing GAMESS and DDI

In Section 2.2.1, the sequence of GAMESS main subroutine is divided into four parts: (1) initializing variables and the communication layer; (2) read in user options; (3) choose a computation branch according to the type of the computation; (4) finalizing memory and communication layer. The right column of Figure 3 shows how we divide and wrap the original sequential main subroutine into several smaller wrapper functions. The wrapper function *gamess_start* is for initializing GAMESS computation and the communication layer (DDI will be initialized); *gamess_end* is for finalizing memory and DDI. The construction of these two wrapper function is simple: basically just wrapping the codes that corresponding to each part and group them into two subroutines. However, parts of the DDI code have to be modified depending on which model is used: the GAMESS/DDI model or the GAMESS/DDI/MPI model.

When using GAMESS/DDI/MPI model, the MPI initialization method *MPI_Init* will be invoked during the initialization of DDI. However, the CCA framework will also call *MPI_Init* at the beginning. Since *MPI_Init* cannot be called more than once, we have to modify DDI to ignore the call to the *MPI_Init* method. A flag is added before the call to *MPI_Init*, so that *MPI_Init* will not be executed if it has already been invoked.

```
int flag;
MPI_Initialized(&flag);
if (!flag) {
  if(MPI_Init(&argc,&argv) != MPI_SUCCESS) {
     fprintf(stdout," DDI: MPI_Init failed.\n");
     fflush(stdout); exit(911);
  }
```

When using the GAMESS/DDI model, the DDI initialization requires a list of command-line arguments, such as process id, port number, hostname, etc, in the DDI-known format, that are passed from the DDI kickoff program. When the GAMESS program works alone (without using the CCA framework), the list of command-line arguments will be passed from the DDI kickoff program to the GAMESS main subroutine, and then passed to the DDI initialization method *DDI_Init*. However, when the GAMESS program works with the CCA framework, the command-line arguments will be passed from the DDI kickoff program to the CCA framework. There is no way that the command-line arguments will be directly passed from the CCA framework to *DDI_Init*.

By using the *StovePipe* library in the CCA framework, the command-line arguments can be read by GAMESS CCA components and then passed to *DDI_init*. Since the *StovePipe* library requires a special format for storing the command-line arguments, such as "*--argument_name*1 *--argument_value*1 ... *--argument_nameN --argument_valueN*", the format of the command-line arguments that created in the DDI kickoff program have to comply with the format that the StovePipe library requires. The format for the arguments will be converted back to the format that DDI knows later by a GAMESS CCA component and be passed to the method *DDI_Init* from the GAMESS CCA component.

Energy, gradient and Hessian calculations. For the third part of the GAMESS main subroutine (the third rectangle from above to the bottom at the left-hand column in Figure 3), several wrapper functions are created: *gamess_get_energy*, *gamess_get_gradient* and *gamess_get_hessian*. This list can be expanded by creating a wrapper function for each

computation type. Those wrapper functions are constructed by setting the "run type" to the corresponding type of computation, such as energy, gradient, Hessian, and optimization. The final results of the computations will be read from the associate common blocks or the direct access files (where GAMESS stores those results).

Initializing the basis set information. GAMESS stores the basis set information, such as primitives, contraction coefficients, and exponents, in the common blocks NSHEL and INFOA (Appendix B has detailed description about the common block NSHEL). A wrapper function has been created for each element in the items of the common blocks.

```
      COMMON /NSHEL / EX(MXGTOT),CS(MXGTOT),CP(MXGTOT),CD(MXGTOT),
     *                CF(MXGTOT),CG(MXGTOT),CH(MXGTOT),CI(MXGTOT),
     *                KSTART(MXSH),KATOM(MXSH),KTYPE(MXSH),KNG(MXSH),
     *                KLOC(MXSH),KMIN(MXSH),KMAX(MXSH),NSHELL

      COMMON /INFOA / NAT,ICH,MUL,NUM,NQMT,NE,NA,NB,
     *                ZAN(MXATM),C(3,MXATM),IAN(MXATM)
```

For example, the *gamess_ex* wrapper function will return the exponent with the specified primitive.

```
/** Get the exponent with the specified index of primitives */
void gamess_ex_(int64_t* index, double* answer);
```

The integral computations

GAMESS computes two kinds of AO integrals, one- and two-electron integrals. For

Table 1. The subroutines for computing integrals

Computation		Subroutine	Description
one-electron integral computation	GAMESS	ONEEI	the driver subroutine for the one-electron integral calculation
		HSANDT	calculate integrals over all shell doublets
	GAMESS Wrapper Functions	gamess_1e_initialize	initialize the one-electron integral calculation
		gamess_dblet_integral	compute integrals for a shell doublet
		gamess_1e_finalize	finalize the one-electron integral calculation
two-electron integral computation	GAMESS	JANDK	the driver subroutine for two-electron calculation
		TWOEI	calculate integrals over all shell quartets
	GAMESS Wrapper Functions	gamess_twoei_initialize	initialize the two-electron integral calculation
		gamess_twoei_compute	compute integrals for a shell quartet
		gamess_twoei_finalize	finalize the two-integral calculation

two-electron integrals, GAMESS provides four computational methods, each of which has its strength for computing different sets of shell types. By default GAMESS chooses the most efficient one by picking the best method for each shell quartet. Users can choose a specific integral code through the input options. For ease of presentation, we omit details of data structures and functions used in integral computations, but list only the driver subroutines for one- and two-electron integral calculations in the GAMESS code and the corresponding wrapper functions in Table 1.

The subroutine ONEEI (Table 1) for the one-electron integral computation in GAMESS is for initializing one-electron integral calculation and calling the subroutine HSANDT to compute one-electron integrals over all pairs of Gaussian shells. A two-level nested loop structure is used in the subroutine HSANDT to loop over all i and j shells, where i and j are indexes of Gaussian shells. However, the *cca-chem-generic* package defines the *compute* method of *IntegralEvaluator2Interface* to return integrals for only one pair of shells; to comply with the interface we cannot just wrap the integral subroutines in GAMESS. In order to create a wrapper function that computes only one shell doublet while making minimum modification to the original GAMESS subroutine, the initialization, finalization, and computation steps are separated into three wrapper functions. Figure 11 shows how we extract the initialization procedure from ONEEI and HSANDT to form a single function for initializing one-election integral calculations. The computation code in HSANDT is wrapped into a function that calculates integrals for one pair of shells with variables (i,j) in the loops as parameters. The wrapper functions are invoked by the *GAMESS.IntegralEvaluator2* (implements *IntegralEvaluator2Interface*) class.

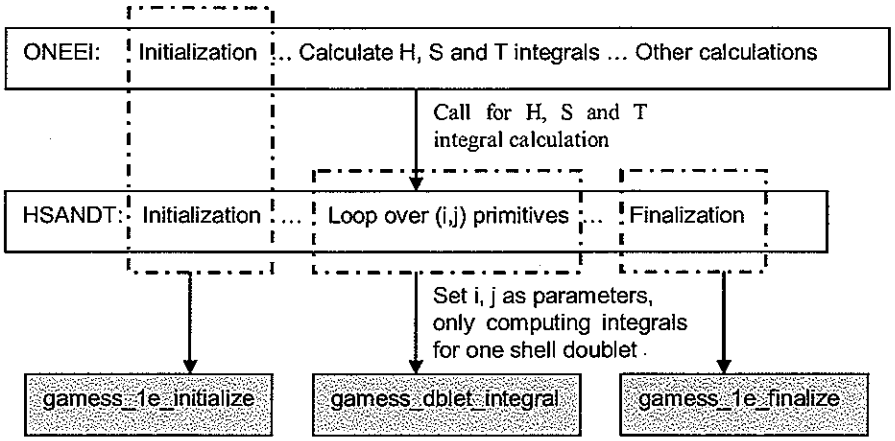

Figure 11. The componentization of one-electron integral calculations in GAMESS.

The subroutine JANDK (Table 1) is the main driver for computing two-electron integrals. It first allocates memory for integral buffers and initializes integral calculations. TWOEI is then called for calculating two-electron integrals over four basis functions. However, the *cca-chem-generic* package defines the compute method of IntegralEvaluator4Interface to return integrals for only one shell quartet. Similarly, we need to create a wrapper function that computes integrals for only one shell quartet.

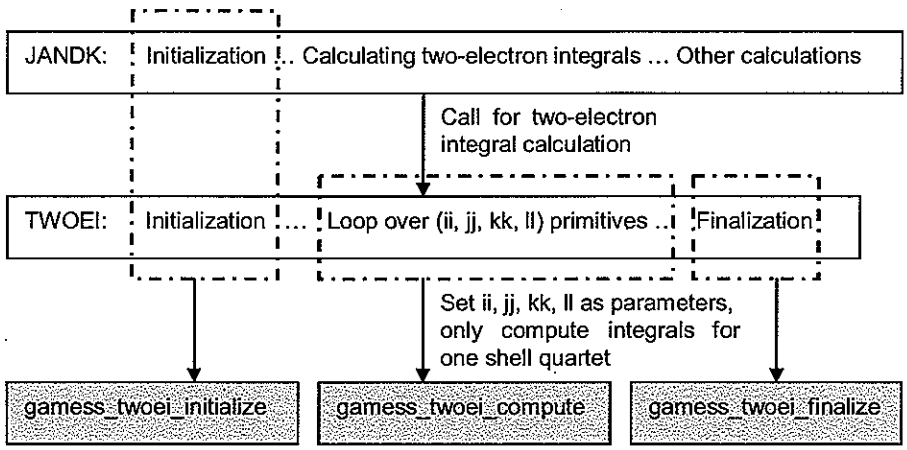

Figure 12. The componentization of one-electron integral calculations in GAMESS

Combining the initialization steps in JANDK and TWOEI (Figure 12), a wrapper function is used for initializing two-electron integrals. With the same strategy as componentizing one-electron integral computations, the part that loops over four basis

functions is wrapped as a function to compute one shell quartet with (*ii*,*jj*,*kk*,*ll*) as parameters. Finally, a wrapper function is created for finalization of two-electron integral calculations.

The reason we separate initialization steps from the computation steps is to reduce the overhead of the wrapper functions. The wrapper functions are designed to compute integrals for a shell doublet or a shell quartet, so they can be called $O(N^2)$ times for one-electron integral calculation and $O(N^4)$ times for two-electron integral calculation. Without separating the initialization step from computation steps, there would be a significant amount of overhead for computing integrals.

3.3.2 The design of GAMESS CCA components

The implementation of GAMESS CCA components is straightforward as long as the associated wrapper functions have been constructed. The *GAMESS.ModelFactory* component implements *ModelFactoryInterface*, and is able to return the *GAMESS.Model* class. The *get_energy*, *get_gradient* and *get_hessian* methods of the *GAMESS.Model* class will invoke the wrapper functions *gamess_get_energy*, *gamess_get_gradient* and *gamess_get_hessian*. Through the *ModelFactoryInterface* Uses/Provides port, the *energy*, *gradient,* and *Hessian* calculations provided by GAMESS can be used through the CCA interfaces. Similarly, the *GAMESS.IntegralEvaluatorFactory* component implements *IntegralEvaluatorFactoryInterface*, and is able to return the *GAMESS.IntegralEvaluator2* and *GAMESS.IntegralEvaluator4* classes for GAMESS integral computations (Figure 6). The *compute* method of the *GAMESS.IntegralEvaluator2* class invokes the wrapper function *gamess_dblet_integral* for computing a shell doublet and the *GAMESS.IntegralEvaluator4* class calls the wrapper function *gamess_twoei_compute* for calculating a shell quartet. Through the *IntegralEvaluatorFactoryInterface* Uses/Provides port, the functionality of the integral calculation can be shared between GAMESS and other chemistry packages.

3.3.3 The structure of GAMESS CCA components

GAMESS stores basis set and molecule coordinates in common blocks, through which the values required for integral computations - the indexes of Gaussian shells, exponents, contraction coefficients, and Cartesian coordinates - are shared among different

subroutines, and integral calculations can be performed. The GAMESS program initializes common blocks, memory, and communications by reading the user input options from an input file. Most of the input options in GAMESS have default values, but the basis set and molecule coordinates are required for all input files. The input file is read for many subroutines during a computation; without this file there is no way GAMESS can be initialized and perform computations. Even though the GAMESS components we developed are based on the interface for a "theoretically independent" component, the underlying wrapper function depends on the original design for initializing the GAMESS computations.

To deal with the common "input file" issue, our approach is to have the *GAMESS.ModelFactory* component create a disk file with the format of the GAMESS input file, based on the user options that are passed from the CCA parameters. This disk file will be passed to the GAMESS wrapper function *gamess_start* to initialize GAMESS computations. Figure 13 shows the dependencies among GAMESS CCA components, GAMESS wrapper functions and the GAMESS program. GAMESS CCA components are built on top of GAMESS wrapper functions, which wrap the functionalities of GAMESS into non-interleaving functions. To construct an application of GAMESS CCA integral computations, a *GAMESS.ModelFactory* component and a *GAMESS.IntegralEvaluatorFactory* component (implements *IntegralEvaluatorFactoryInterface*) are instantiated in a CCA framework. This framework is middleware implementing a CCA model [14]. The *GAMESS.ModelFactory* component reads user input options from CCA parameters, creates a GAMESS input file on disk based on those input options and calls the wrapper function *gamess_start* to read the input file and initialize GAMESS common blocks and communications. The *GAMESS.ModelFactory* component also provides a *GAMESS.Model* class (implements *ModelInterface*) for calculating the energy, gradient and Hessian. After GAMESS computations are initialized successfully, the GAMESS.IntegralEvaluatorFactory component is able to provide the *GAMESS.IntegralEvaluator2* class (implements *IntegralEvaluator2Interface*) and the *GAMESS.IntegralEvaluator4* class.

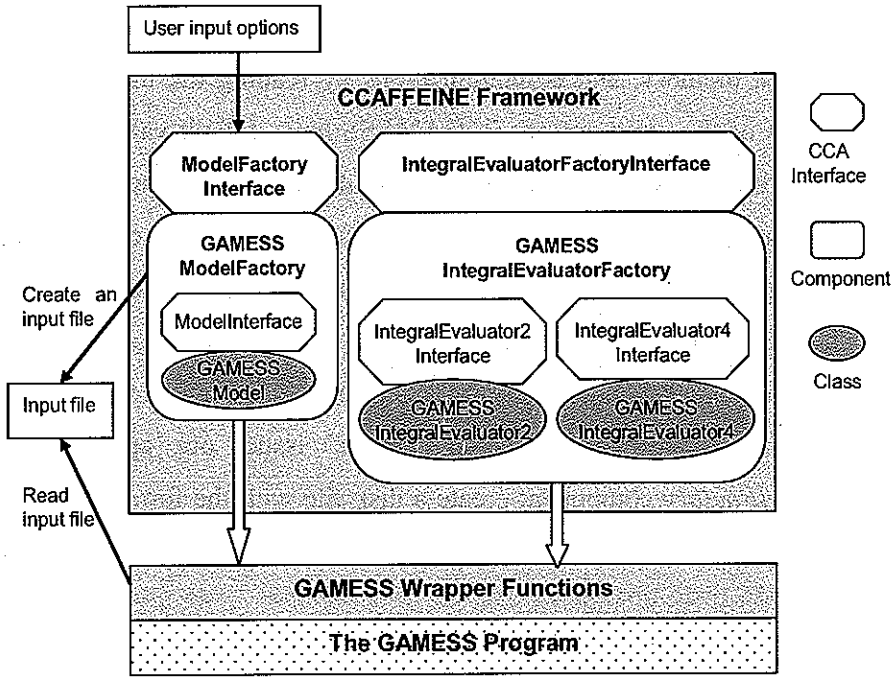

Figure 13. GAMESS CCA components are built on top of GAMESS wrapper functions. However, the initialization of GAMESS computations could not be componentized and relies on an input file for initializing common blocks and communications.

CHAPTER 4. INTEGRATION

The purpose of this research is to solve the interoperability of the three chemistry packages: GAMESS, NWChem and MPQC (more packages may be involved in the future), to share the functionalities among those packages. Through the pre-defined CCA chemistry interfaces, a package is able use the functionalities provided in other packages under the CCA framework. This resource sharing enables a new computation being constructed quickly by choosing components from one or several preferred packages.

However, the integration of components from the existing packages is not as easy as integrating components that are created from scratch. The components from GAMESS and NWChem are based on the large legacy codes that are mostly written in FORTRAN 77. The functionalities, parallel mechanisms and underlying structures of those components are restricted by the design of the existing legacy codes. Even for the components that perform the same kinds of computations but from different packages, the way of using those components may be different. For example, the two-electron integral computations in GAMESS are implemented with a load balancing mechanism that allows the tasks (integrals) distributed among processes evenly. Instead of using the load balance mechanism to parallelize the integral computation itself, both NWChem and MPQC parallelize the routines that call the integral computations. This makes the way of using CCA integral components from GAMESS different from the components provided by the other two packages.

Theoretically, users should not worry about the underlying design of components. However, especially for the components constructed from the large legacy code, this is hard to achieve in practices. The well-designed interfaces and the set of fully tested components may help us to create a user-friendly, flexible, and powerful component-based software system for the quantum chemistry simulations.

As a starting point for integrating the three chemistry packages, we choose to integrate the integral calculations. We use the *GAMESS.ModelFactory* component for reading user input options; pass a *GAMESS.GaussianBasisMolecular* object to the *MPQC.IntegralEvaluatorFactory* component; calculate integrals by using the integral evaluators from MPQC. Since the CCA integral components for NWChem were still under

development at the time we constructed the integration steps, we will leave the integration with NWChem as one of our future works.

To generalize the integration of the already componentized computations, such as energy, gradient, Hessian, and integral, we designed an interface for the "client-side" functions of GAMESS CCA components. The "client-side" in this research is a set of classes and data structures that are designed and created by using the CCA chemistry classes/components with specific language binding. The programmers can choose a language binding from the list that allowed by BABEL, such as C/C++, FORTRAN 77/90, Java, and Python. The corresponding language bindings for a component can be generated by using the BABEL tools. For example, the *GAMESS.ModelFactory* component is implemented in C++. If we need to create the "client-side" for GAMESS CCA components with FORTRAN 77, the "f77" binding has to be generated before the methods in the *GAMESS.ModelFactory* component can be accessed from the FORTRAN 77 "client-side" functions.

The section 4.1 will show the integration steps for integral calculations by GAMESS and MPQC CCA components. The section 4.2 will introduce the design mechanism of the GAMESS client-side and the possible issues for implementing the client-side interfaces.

4.1 The Integration of the Integral Calculation

We have already introduced the implementation details of GAMESS CCA integral components and the structure of using GAMESS CCA components. To demonstrate the procedure of integrating the integral calculation over the three chemistry packages, we need to have an overall knowledge of the CCA integral components from both MPQC and NWChem. Then, the procedure of integrating GAMESS and MPQC to perform the two-electron integral computation will be presented.

MPQC CCA integral components

MPQC components are derived in a straightforward manner from the class libraries underlying the MPQC package. For example, the *IntegralEvaluator4* CCA object simply wraps a class derived from *sc::TwoBodyInt*. On the client side, CCA integral factories are wrapped by the *sc::IntegralCCA* class and CCA evaluators, such as *IntegralEvaluator4*, are wrapped by the appropriate evaluator class, such as *sc::TwoBodyIntCCA*. Thus, MPQC has

no code that directly uses CCA integral interfaces, with all function calls to CCA objects occurring through a wrapper object implementing an abstract interface. There are two integral evaluator factories available within MPQC, *IntV3EvaluatorFactory* and *CintsEvaluatorFactory*, providing access to the native *IntV3* integral package and the *Libint* package [15]. Details about the design of MPQC integral components are described in a previous publication [16].

NWChem CCA integral components

As with the GAMESS code, the NWChem component software essentially consists of wrappers to access the capabilities of the NWChem integral API. Currently, the *NWChem.ModelFactory* needs to be created and initialized so that NWChem has the proper information concerning the basis sets and the molecular configuration. It is anticipated that this will change in the future. Once the Model Factory has created a Model, then NWChem has also initiated its other functionalities such as memory management (global array allocation), communication protocols and run-time database management. This is currently essential for the integral components to function properly.

A significant portion of the CCA integral interface is similar to the NWChem API and there is a fairly direct one-to-one mapping. However, the *IntegralDescrInterface* is significantly different with no analog in NWChem, so the specifics of the types of computations that the API is to perform are kept in the components and translated to the appropriate API calls.

The integral termination is straightforward. However, the appropriate Model also needs to be terminated to end all of the NWChem processes. Since NWChem CCA components are currently being upgraded from working with the older version of Babel tools and the CCA framework to working with the newest version of those packages, the integration of GAMESS and NWChem will be part of our future work.

Interoperability between GAMESS and MPQC

To test interoperability between packages, we pass the basis set information, the type of integrals, and molecule coordinates from a *GAMESS.ModelFactory* component to a MPQC *integral evaluator factory* component by invoking a *get_evaluator* method. For

example, the SIDL definition for the *get_evaluator4* method of *IntegralEvaluatorFactoryInterface* is showed as follows:

```
/** Get a 4-center integral evaluator
    @param desc Integral set descriptor
    @return 4-center integral evaluator */
IntegralEvaluator4Interface get_evaluator4(
     in CompositeIntegralDescrInterface desc,
     in MolecularInterface bs1,
     in MolecularInterface bs2,
     in MolecularInterface bs3,
     in MolecularInterface bs4);
```

Using MPQC *integral evaluators* is expected to be as straight forward as using GAMESS *integral evaluators*, as long as everything is initialized properly. For example, our current testing is to pass a *GAMESS.GaussianBasisMolecular* object to the *MPQC.IntV3EvaluatorFactory* component through the *IntegralEvaluatorFactoryInterface* provides/uses connection. If the initialization in the *GAMESS.GaussianBasisMolecular* object is correct, then the *MPQC.IntV3EvaluatorFactory* component should be able to return an *integral evaluator* and do the same computation as a GAMESS *integral evaluator*.

The integration steps are as follows:

(1) Instantiate a *GAMESS.ModelFactory* component and a *MPQC.IntV3EvaluatorFactory* component in a CCAFFEINE framework.

(2) *GAMESS.ModelFactory* component reads user options through CCA parameters and initializes GAMESS common blocks, memory and parallel layers.

(3) Create a *GAMESS.GaussianBasisMolecular* object and a *CompositeIntegralDescr* (implemented by the ***cca-chem-generic*** package) object.

(4) Pass the *GAMESS.GaussianBasisMolecular* and *CompositeIntegralDescr* objects to the *MPQC.IntV3EvaluatorFactory* component and get the reference to a *MPQC.IntegralEvaluator2* object.

(5) Invoke the *compute* method of the *MPQC.IntegralEvaluator2* object inside a two-level loop structure that computes integrals over all pairs of shell basis functions.

```
for (int64_t ii=0; ii<nshell; ii++) {
      for (int64_t jj=0; jj<=ii; jj++) {
          eval2_.compute(ii,jj);
      }
}
```

(6) Pass the *GAMESS.GaussianBasisMolecular* and *CompositeIntegralDescr* objects to the *MPQC.IntV3EvaluatorFactory* component and get the reference to a *MPQC.IntegralEvaluator4* object.

(7) Invoke the *compute* method of the *MPQC.IntegralEvaluator4* object inside a four-level loop structure that computes integrals over all shell quartets.

```
for (int64_t ii=0; ii<nshell; ii++) {
     for (int64_t jj=0; jj<=ii; jj++) {
         for (int64_t kk=0; kk<=jj; kk++) {
             for (int64_t ll=0; ll<=(kk==ii?jj:kk); ll++) {
                 eval4_.compute(ii,jj,kk,ll);
             }
         }
     }
}
```

(8) Finalize and remove all objects and components.

The goal of this experiment is to test interoperability only. The results of an integral computation in each iterate are usually used by some other computation. With initial interoperability established, our future work will turn to componentizing GAMESS code that utilizes GAMESS/MPQC/NWChem integral components. The performance of GAMESS integral components and issues in the interoperability of GAMESS with MPQC integral components are discussed in Chapter 5.

4.2 The Design of the GAMESS Client-Side

We have showed the preliminary experiments on integral calculations by using CCA components provided by GAMESS and MPQC. In this experiment, we instantiate a *Chemistry.MoleculeFactory* component, a *GAMESS.ModelFactory* component, a *MPQC.IntegralEvaluatorFactory* component, and a *driver* component in the CCA framework and several auxiliary classes have also been created, such as *GAMESS.GaussianBasisMolecular*, MPQC.*IntegralEvaluator2*, *MPQC.IntegralEvaluator4*, and *Chemistry.Molecule* classes. The user input options are read by the *GAMESS.ModelFactory* component; the basis set information and molecular geometry are

stored in a *GAMESS.GaussianBasisMolecular* object, which is passed to the *MPQC.IntegralEvaluatorFactory* component; and the one- and two-electron integrals are calculated by MPQC integral evaluators.

However, this experiment is just for calculating all shell doublets and shell quartets for a molecule. A *driver* component is needed to manage the procedure of the computation. Whenever a new computation is required, or a new package joins in, some modification has to be done in the *driver* component or the SIDL interfaces, etc. For example, if we want to use CCA integral components provided by NWChem, a different loop structure may be used instead of the loop structures we listed above for looping over all shell multiplets. Or if we need to construct an energy calculation by using the integrals calculated by a CCA component, an option may be added: choose a program that will be used to calculate the integrals from the list {GAMESS, MPQC, NWChem}. There may be other options or SIDL interfaces required to construct a computation, which will complicate the implementation of each component.

A more flexible way of implementing a computation of multiple packages through components is to wrap the functionalities implemented for components to create the object-oriented client-side classes. In this section, the C++ client-side interfaces for the GAMESS computations, such as energy, gradient, and Hessian, will be presented, by using integrals calculated by integral evaluators from GAMESS, NWChem or MPQC. Before we jump into the detailed design, we need to understand how such a computation is processed. The sequential steps for performing an energy calculation are as the follows:

1. Initialize GAMESS computations from a GAMESS input file: create a *GAMESS.Model* object, from which the *gamess_start* and *gamess_read_input* subroutines are invoked.
2. Create a *GAMESS.GaussianBasisMolecular* object based on the basis set and molecular geometry information from the GAMESS input.
3. Create an *IntegralEvaluatorFactory* object for the specified package (GAMESS, MPQC or NWChem); pass the *GAMESS.GaussianBasisMolecular* object and a *ChemistryIntegralDescrCXX.CompositeIntegralDescr* object as parameters to get an integral evaluator (1-, 2-, 3-, or 4-center).

4. Call the *get_energy* function of the *GAMESS.Model* object and the underlying integral calculations are performed by using the integral evaluators from step 3.

Several Issues for Designing the Client-Side Interface

There are several issues we have to take care in the design of such a client-side interface. For different chemistry programs, different loop structures for looping over all multiplets are used (e.g. GAMESS uses a different 4-level loop structure for 2e-integral computation from the one MPQC uses). The appropriate loop structure should be chosen for the specified package as long as the end user picks a package for doing the integral computations.

Also, the integral ordering in GAMESS is different from the integral ordering in MPQC and NWChem (these two programs use the integral ordering defined by the cca-chemistry group [24]). The conversion of the integral ordering for the integrals of each shell multiplet should be done automatically before the integrals being used in a computation. Since both MPQC and NWChem use the integral ordering defined in Joe Kenny's paper [24], we only need two kinds of conversion: from the integral order used in GAMESS to the integral orders used in MPQC & NWChem; from the integral orders used in MPQC & NWChem to the integral orders in GAMESS. These two kinds of conversion must be incorporated within the client-side design of GAMESS.

Finally, some language interoperability issues need to be considered carefully when constructing the client-side implementations. The underlying GAMESS computations are implemented in FORTRAN 77, and the integral computations of MPQC is implemented in C++. When constructing the C++ client-side of the GAMESS program, we should avoid directly calling the GAMESS wrapper functions, instead those function calls should be hidden in the server-side implementations.

For example, when calling a wrapper function that takes a parameter of the type "int64_t" from the C++ client-side, I got a bunch of errors that "int64_t" is not defined. However, if an integer of the type "int64_t" is defined inside the C++ client-side code, not being passed to GAMESS wrapper functions, I will not get any errors. On the other hand, if the same wrapper function is invoked through the server-side implementations, the same errors will occur.

The reasons for designing the client-side in C++. It is natural to create the object-oriented design by programming in C++. The real computations are performed by the wrapper functions and GAMESS program, and the C++ client-side is used for reading user input options and facilitating corresponding configurations, such as which package will be used for providing integral evaluators. When only the references to integral evaluators are passed from C++ to FORTRAN 77 for performing integral calculations for GAMESS computations, the performance overhead from the language interoperability should not be large. Since the C++ client-side should be easier to implement than the FORTRAN client-side for GAMESS, it could be an experiment for implementing the FORTRAN client-side.

The Design of the C++ Client-Side Interface

Basically, several classes are designed for wrapping the integral computations provided by CCA chemistry components: *ClientIntEvalFactory (*wraps *IntegralEvaluatorFactory* components*)*, *ClientInteEval1* (wraps *IntegralEvaluator1* class), *ClientIntEval2* (wraps *IntegralEvalutor2* class), *ClientIntEval3* (wraps *IntegralEvaluator3* class) and *ClientIntEval4* (wraps *IntegralEvaluator4* class). For each class, there is a field: *package_*, for specifying the name of the underlying program. The class *GAMESSCCAComputation* is designed for GAMESS to perform chemistry computations with the references to a *GAMESS.Model* object and a *ClientIntEvalFactory* object from the specified program.

GAMESS iteratively calculates and stores the integrals for a shell multiplet in a one-dimensional array. The integrals in this array will be either written to a disk file (the *conventional* method) or immediately used by other subroutines (the *direct* method). The integral evaluators from MPQC or NWChem will return a SIDL array of double data type for the integrals of specified shell multiplet. The SIDL array returned by those integral evaluators can be converted to a one-dimensional array and passed to GAMESS through the CCA interfaces. We need to make sure the ordering of integrals in the array is the same as the ordering in GAMESS integral array. A GAMESS wrapper function *gamess_reorder*, for converting the integrals with the orders used in MPQC/NWChem to the orders used in GAMESS, is needed before any integrals be used in GAMESS computations. On the other

hand, a method *reorder_gtom* in a *ClientIntEval* class is also needed to convert the integral ordering in a GAMESS array to the format that MPQC & NWChem use.

In addition, several FORTRAN 77 functions are needed for underlying integral computations. For example, *gamess_eval2* is designed to loop over all 2-center integrals by using the integral evaluator passed from the C++ interface (from MPQC or NWCHEM), where the memory address of the integral evaluator2 is passed as "INTEGER*8". Similarly, the function *gamess_eval4* is designed for looping over all shell quartets. Figure 14 show the structure of the GAMESS client-side interfaces for computing energy, gradient and Hessian.

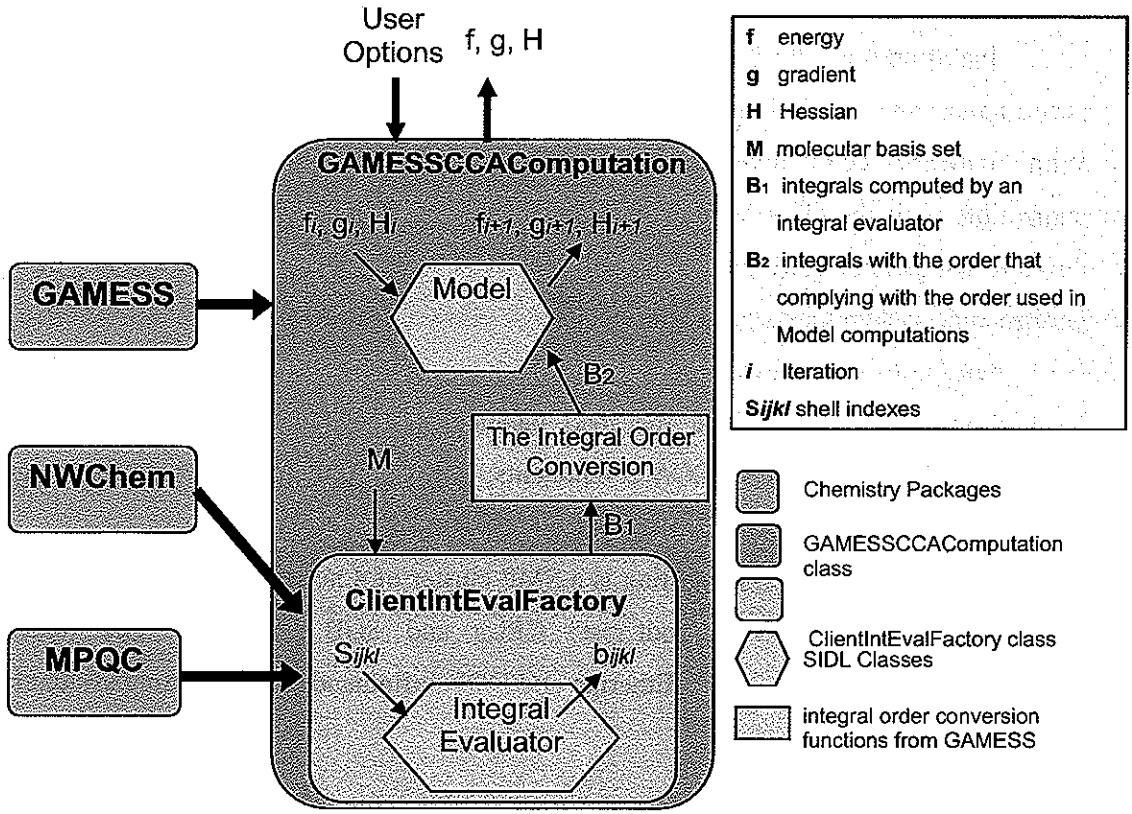

Figure 14. The Client-Side design for GAMESS computations. A GAMESS.Model class is used for performing energy, gradient and Hessian calculations. The underlying integral calculations are provided by one of the three chemistry programs: GAMESS, NWChem and MPQC. For the integrals provided by MPQC and NWChem, the integral orders will be converted to the orders used in GAMESS before they are used in any GAMESS computations.

CHAPTER 5. PERFORMANCE EVALUATION

Within the scope of GAMESS, performance bottlenecks can occur in many places such as cache utilization, I/O or communication. Performance evaluation and monitoring tools for each of these potential bottlenecks may take years to develop, so starting from scratch is not a feasible solution. A useful approach is to use existing performance tools such as TAU (Tuning and Analysis Utilities) [29] or PAPI [30I], and incorporate them into GAMESS. These performance tools usually provide APIs for application developers to develop performance evaluation functions according to application needs.

Incorporating performance tools into GAMESS usually requires inserting performance function calls into the GAMESS source code, which is an intrusive approach. With GAMESS components, we prefer a performance tool that provides an interface compatible with the CCA standard, such that the access to performance tool APIs can be through component ports instead of direct calls to the API. In particular, the TAU performance system meets our requirements.

Our performance evaluation includes three parts: (1) test the overhead incurred by the CCA framework; (2) evaluate the load balance strategy for the two-electron integral calculation used in GAMESS CCA components; (3) explore the performance for integrating the integral calculation of GAEMSS and MPQC.

The platform used for testing is a SMP cluster of 4 nodes, where each node has two dual-core 2.0GHZ Xeon "WoodCrest" CPUs and 8GB of RAM. The nodes are interconnected with both Gigabit Ethernet and DDR Infiniband. The operating system is Red Hat Enterprise Linux 4.

5.1 TAU Performance Tools

TAU is based on a general computation model [29], which is a superset of the one used by GAMESS. It provides technology for performance instrumentation, measurement, and analysis for complex parallel systems. Performance information can be captured at the node/context/thread level by using TAU. Besides performance instrumentation capability on both the component level [31] and the source code level, TAU also provides an interface to access the hardware counters through PAPI or PCL [31].

For CCA applications, TAU provides a performance component to measure the performance of CCA component software through the common *MeasurementPort* interface. Besides the performance component, TAU also provides *MasterMind* and *Optimizer* components for performance data collection for performance modeling of components and constructs optimal component assemblies, and Proxy Generators build proxies for both the *MeasurementPort* and the *Monitorport* in performance component [32]. To successfully install the TAU performance component and use all the provided functionality, both TAU and PDT (Program Database Toolkit) [33] must first be installed TAU performance components then can be set up.

5.2 Test the Performance Overhead of the CCA Framework

To test the overhead of the CCA framework in GAMESS calculations, we compared the wall-clock times (in seconds) of the RHF energy calculations for four molecules: ergosterol, Darvon, luciferin and nicotine, by using GAMESS with and without the CCA framework. In both cases, the GAMESS/DDI/MPI model will be used, since this is the model we will use for GAMESS to integrate with other packages through components. The TAU timer is inserted between the calls to calculate energy in the GAMESS program and the *get_energy* method of the *GAMESS.Model* class.

First, all the computations will run in sequential for testing the overhead incurred by the CCA framework in a single CPU. Table 2 shows the wall-clock time of the energy computations by using the GAMESS program (the second column) and the *GAMESS.ModelFactory* component (the third column). For the GAMESS program, the type of the computation is set as "energy" in the user input file. For the *GAMESS.ModelFactory* component, the *get_energy* method of a *GAMESS.Model* class is called. The results show that

Table 2. The wall-clock time (Seconds) for the RHF energy calculation with & without the CCA framework

Molecule	No CCA	With CCA
Darvon	3602.3	3607.4
luciferin	138.2	143.3
nicotine	61.9	64.3
h2o (CCQ)	10.1	11.2

the performance overhead incurred by using the CCA framework is less than 10~15 percent of the wall-clock time when using the GAMESS program without CCA.

Then we run the energy calculation of the molecule "nicotine" in parallel for comparing the scalability of the GAMESS program with and without CCA. Figure 15 shows that the scalability of the GAMESS program is similar as the scalability of the GAMESS CCA components.

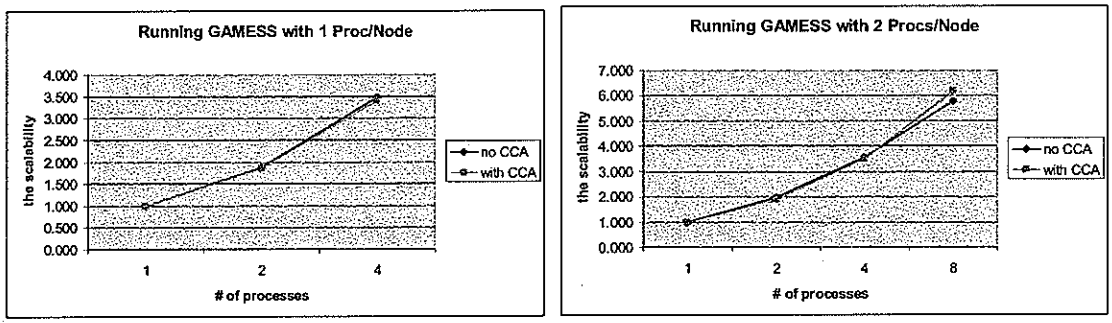

Figure 15. The energy calculation of the molecule "nicotine" run on both the original GAMESS program and the GAMESS CCA component, which we labeled as "no CCA" and "with CCA", respectively.

5.3 The Load Balance in Two-Electron Integral Computations

There are two kinds of load balancing strategies used in GAMESS to distribute the tasks of calculating two-electron integrals among processes: the dynamic load balance and the static load balance. For the dynamic load balance strategy, the tasks are dynamically assigned to a process and a global counter in DDI is used to make sure each task will be executed exactly once. This method adjusts the distribution of the tasks among processes dynamically, since the current CPU usages and the quality of the network connection will both affect the results whether or not a task is assigned to a process. For the static load balance strategy, the tasks are assigned to each process according to the identity of the process. This method guarantees the number of the tasks assigned to each process is the same. Theoretically, the static load balance is more stable since the number of tasks assigned to every process is the same, and the dynamic load balance is more efficient since the number of tasks will be adjusted dynamically according to the work load of a process. By default, the dynamic load balance is normally used in GAMESS.

```
DO 920  II = IST, NSHELL  (first level)
    DO 900  JJ = JST, II  (second level)
        IF USE DYNAMIC LOADBALANCE, GET THE CURRENT GLOBAL COUNTER
        AND DECIDE IF CONTINUE WITH THE INNER BLOCK OF THE LOOP.
        DO 880  KK = KST, JJ  (third level)
            IF USE STATIC LOADBALANCE, THE ID OF THE CURRENT PROCESS
            COULD DECIDE IF CONTINUE WITH THE INNER BLOCK OF THE LOOP.
            DO 860  LL = LST, KK  (forth level)
                CHECK FOR REDUNDANTIES BETWEEN THE 3 COMBINATIONS
                (II,JJ//KK,LL), (II,KK//JJ,LL),(II,LL//JJ,KK)

                COMPUTE SHELL QUARTET AND PROCESSING THE RESULTS

            860 CONTINUE
        860 CONTINUE
    860 CONTINUE
860 CONTINUE
```

Figure 16. Load balancing in GAMESS TWOEI subroutine. The small case letters inside parenthesizes indicate the level number of each loop. The block of inner loops surrounded by the solid line shows the size of the task for a dynamic loading procedure. The block of inner loops surrounded by the dashed line shows the size of task for a static loading.

In GAMESS the two-electron integrals are computed inside a 4-level nested loop structure (Figure 16) in the TWOEI subroutine, The dynamic load balance is putted after the second level of the loop, where the size of a task for each dynamic loading procedure is the block of inner loops surrounded by the solid line. A global counter decides if a process needs to continue with the inner loops for each loading procedure and each task is performed by exactly one process. The static load balance is arranged after the third level of the loop, such that the size of the task for each loading procedure is the inner loops surrounded by the dashed line. Since the index of each task and the id entity of a process decide if the process continues with the inner loop, no communication is needed in the static load balancing.

However, the chemistry integral interface *IntegralEvaluator4Interface* defines the *compute* method to compute one shell quartet at a time. When we keep the load balance being handled in the wrapper function - *gamess_twoei_compute* (this wrapper function called

by *compute* method of *IntegralEvaluator4*), the size of the task for each loading procedure is just a single shell quartet. This is analogues to move the load balance in TWOEI to the 4th level of the loop, and there is no guarantee that the performance in the original GAMESS 2e-integral computation would be preserved.

We will use the molecule "nicotine" to test the performance and scalability of the two-electron integral calculation when the dynamic load balance is located after the 2nd level, the 3rd level and 4th level of the loop structure. Three groups of the performance data will be compared and the results will help us to find an appropriate strategy to move the load balance of the two-electron integral calculation from GAMESS to the component level without sacrifice the performance. Note that we will not show the performance of the two-electron integral calculation in TWOEI when moving the static load balance to the 4th level of the loop structure since the change of the performance is not significant.

Test dynamic load balance. We run the 2e-integral calculation of "nicotine" by using 1.1, 2.1, and 4.1 nodes in GAMESS/DDI/MPI mode, **where *x.y* means in that experiment we use *x* nodes and *y* CPUs on each node**, and compare the scalability show the wall-clock time for each node when using dynamic load balance in 2nd, 3rd and 4th level of the loop structure. The upper chart of Figure 16 shows that the performance is much worse when moving the dynamic load balance to the 4th level of the loop structure. When using 4 processes, the wall-clock time of calculating 2e-integrals when the dynamic load balance is at the 4th level is almost double the wall-clock time when the dynamic load balance is at the 2nd or the 3rd level. The lower chart of Figure 17 shows that the tasks are distributed unevenly among processes when using 2.1 or 4.1 nodes, where in each case the process 0 computes almost all of the shell quartets. This also causes the poor scalability when running in more than one node.

The load balance in GAMESS CCA integral components

From the performance results showed in Figure 16, when the load balance is handled in the 4th level of the loop structure, the number of tasks will be distributed unevenly among processes when using more than one node, which will also lead to the poor scalability. This means that we should not reduce the size of a task to a shell quartet. Since each function call to the *compute* method of an integral evaluator4 returns the integrals of a shell quartet, we

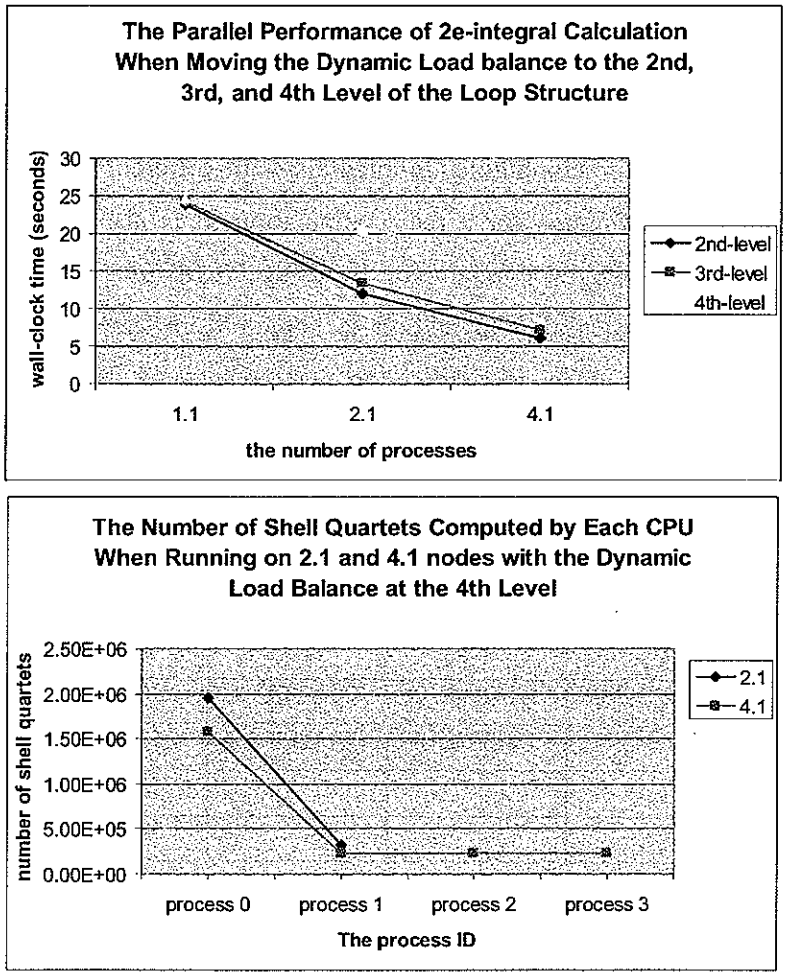

Figure 17. The upper chart shows the parallel performance of the 2e-integral calculation when moving the dynamic load balance to the 2nd, 3rd, and 4th level of the loop structure. The lower chart shows the number of shell quartets computed by each process when the dynamic load balance is moved to the 4th level of the loop structure.

cannot handle the load balance inside the wrapper function, or inside an integral evaluator. Not losing or limiting the functionalities of the original GAMESS program, we copy the loop structure for the 2e-integral calculation in the original GAMESS code to a *TWOEIDriver* component that use the same load balancing approach as TWOEI, except that the 2e-integrals are calculated by calling the *compute* method of the *GAMESS.IntegralEvaluator4* object.

The implementation of *TWOEIDriver* has a flag – "load_balance" for choosing a type of the load balance for the 2e-integral computation from components. The available options are:

```
load_balance = 0, if no load balance
load_balance = 1, if static load balancing is used
load_balance = 2, if dynamic load balancing is used [DEFAULT]
```

The *TWOEIDriver* component is presented as an example of using GAMESS CCA integral components for computing 2e-integrals with different choices of load balancing methods, not being designed for a real computation. It is also used for the performance evaluation of the 2e-integral computation by using GAMESS components. Since the loop structure for the 2e-integral computation is copied from the TWOEI subroutine to the component-level, it is fairly to predict the performance of the static and dynamic load balance of the 2e-integral computation by using GAMESS CCA components should be as good as the performance by using GAMESS.

Table 3. The number of shell quartets computed by using the dynamic load balance strategy in the *TWOEIDriver* component

	1.1	1.2	2.1	1.4	2.2	4.1
process 0	2.2808E+06	1.16488E+06	1.11668E+06	567397	571247	555183
process 1		1.11592E+06	1.16413E+06	576551	565618	570469
process 2				557617	580110	564786
process 3				579240	563830	590367

Table 3 shows the number of shell quartets computed by each process when performing 2e-integral computation with GAMESS CCA components, where the number of shell quartets computed on each process is very close when using 1.1, 1.2, 2.1, 1.4, 2.2 and 4.1 nodes.

5.4 Performance Evaluation for Integral Computations

In this section we present only the performance of the two-electron integral computation since this computation takes significantly more CPU time than the one-electron integral computation does. We measure the wall-clock time for calculating all shell quartets of a molecule by using the GAMESS program, GAMESS wrapper functions, GAMESS CCA integral components and GAMESS & MPQC CCA components. First, we examine the performance overhead incurred by the design of the wrapper functions. This is done by

invoking the *gamess_twoei_compute* wrapper function inside the four-level nested loop structure, and comparing the results with the time of the same computation by using the original GAMESS two-electron integral computations. Second, we examine the performance overhead caused by the CCAFFEINE framework when running the GAMESS CCA integral computations. This is done by evaluating the performance overhead of *GAMESS.IntegralEvaluator4* class, which in turn uses the wrapper functions for calculation. Finally, we examine the performance overhead incurred by the integration of GAMESS and MPQC.

The TAU performance tools are used for measuring the performance of two-electron integral computations. We insert TAU timers in both component-level methods and in GAMESS subroutines. The wall-clock time of looping over all shell quartets is used as the performance data and the time is measured in seconds.

Since both NWChem and MPQC parallelize the routines that call the integral computations, instead of parallelizing the integral computations themselves, we have decided to show only sequential performance data here. .

Test cases. Four molecules are used as our test cases. Table 4 shows the names of the molecules, the basis set, the number of atoms, the number of shells, the number of basis functions, and the number of shell quartets. The test cases are listed in descending order according to the number of two electron integrals.

The integral screening in GAMESS two-electron integral computation. *Integral screening* is a technique to ignore calculating integrals which are estimated to have little or no contribution to the final results of the Fock matrix [22]. GAMESS by default uses integral screening techniques to screen out small integrals in the two-electron integral computation. In the design of CCA integral components, the integral screening has been separated from the integral computation, and is used as an independent option. Since the three chemistry packages use different screening techniques and default thresholds for small integrals, the number of non-zero two-electron integrals being calculated by each package is different from each other. We turn off the integral screening in every package when conducting interoperability testing to make sure every integral component will compute the same number of shell quartets.

Table 4. Test GAMESS integral computation

molecule	basis set	# of atoms	# of shells	# of basis functions	# of shell quartets
ergosterol	6-31G*	73	204	523	2.18625E+08
Darvon	6-31G*	54	158	433	7.88956E+07
luciferin	6-31G*	26	90	294	8.38656E+06
nicotine	6-31G*	44	76	208	4.2822E+06

In GAMESS, a native buffer (in memory), GHONDO, is allocated for storing 2e-integrals of one shell quartet. The results of GHONDO are either read and saved to a disk file, or used immediately, and the values of GHONDO are reset to zeros and used for storing 2e-integrals for another shell quartet in the next iteration. However, to componentize 2e-integral calculations for a shell quartet, the results should be stored in a buffer passed from a calling function (or an *integral evaluator4*). Instead of using GHONDO for storing the results of computing a shell quartet, we use the buffer passed to the wrapper function. The resulting integrals of each shell quartet can be accessed through the reference to the buffer by the end of each iterate and no disk I/O is needed for writing the results to a disk file.

To compare the performance of the original GAMESS subroutine and the wrapper function, we modified the original GAMESS code to ignore disk I/O after computing each shell quartet (to be compatible with our design in the wrapper function). The second column of Table 5 shows the performance data for computing 2e-integrals in GAMESS.

Test GAMESS wrapper integral computation. The third column of Table 5 shows the performance for 2e-integral computation using wrapper functions. The overhead of the 2e-interal computation using the wrapper functions is about 17% of the 2e-integral computation with the original GAMESS code.

In the original GAMESS code, two-electron integrals are computed by looping over all shell quartets in four nested loops. In GAMESS wrapper functions, the *gamess_twoei_compute* function computes one shell quartet at a time. Thus, when looping over all shell quartets, we have $o(N^4)$ function calls to the *gamess_twoei_compute* function. In the original GAMESS code, statements that are inside the first, second or third-level of the four-level loop structure, now need to be executed for each shell quartet, about $o(N^4)$ times. If

Table 5. Wall-clock times (sec) for two-electron integral computations

molecule	GAMESS	GAMESS Wrapper Functions	GAMESS CCA Components
ergosterol	801.52	921.35	980.16
Darvon	361.47	422.72	445.15
Luciferin	63.39	74.11	77.06
Nicotine	22.93	26.71	28.50

there is an overhead introduced by each single call to the compute method, the overall performance overhead can be significant.

Test GAMESS CCA integral computation. The goal of this experiment is to test the performance overhead of the CCAFFEINE framework. The GAMESS wrapper functions are used for implementing GAMESS CCA components. Thus, a buffer is passed from a *GAMESS.IntegralEvaluator4* object to the GAMESS wrapper functions for storing results of a shell quartet and the reference to the buffer is returned. The fourth column of Table 5 shows the running time of the 2e-integral calculation obtained using GAMESS CCA integral components. It shows that the performance overhead is relatively small, since all times are within 10% of the original running time. The same amount of performance overhead incurred by the CCA framework has also been mentioned in the previous literatures. However, the total performance overhead incurred for componentizing integral calculation (include the wrapper functions and CCA frameworks) is relatively large, about 28.7% (1.17*1.1-1). This overhead may be reduced through either implementing GAMESS CCA components in FORTRAN (the current version is implemented in C++), or refining the GAMESS wrapper functions.

Integration of GAMESS & MPQC. Integral computations using CCA components from both MPQC and GAMESS are conducted through the process outlined in Section 3.5. In our testing, we produced the wall-clock time for computing two-electron integrals by using GAMESS CCA components, and GAMESS & MPQC components. Here we choose the water molecule with the cc-pVQZ and aug-cc-pVQZ basis sets to perform the two-electron integral calculations. The performance results of such two-electron integral calculations by using the original GAMESS program and the original MPQC program are expected to be very close, since the water molecule is relatively small and the basis sets we

used here is quite large. The MPQC program contains only one integral code, which is sophisticated and slower than some integral codes in GAMESS (there are four different integral codes in GAMESS). When using large basis sets, GAMESS will choose the more sophisticated/slower integral code, which has similar performance with the integral code in MPQC. Table 6 shows that the discrepancy of the 2e-integral computation for the water molecule is very small between GAMESS CCA components and GAMESS & MPQC CCA components, and these results are exactly what we have expected.

Table 6. Wall-clock times (sec) for testing the water molecule with GAMESS and MPQC

basis set	GAMESS CCA Components	GAMESS & MPQC CCA Components
cc-pVQZ	3.63	3.65
aug-cc-pVQZ	16.07	15.96

CHAPTER 6. DISCUSSION AND CONCLUSION

In the process of developing integral components, several issues affected our design of components, or delayed the progress of component development. We discuss these issues in this section.

Low-level interoperability

Ideally if similar functions from different packages are componentized, complying with the same interface, we should be able to use these components interchangeably. However, if components are designed without substantial modifications to existing applications (e.g., using wrapper functions), the ``plug-and-play" goal may be difficult to achieve.

The differences in the approaches to develop integral components provide a good example of the difficulties faced in interfacing low-level components in a "plug-and-play" fashion. For the MPQC integral component, the underlying software architecture is object-oriented and is more amenable to the encapsulation concepts of component architectures. For GAMESS, a package with over two decades of development history and developers scattered around the world, encapsulation into components may be error-prone in part because the subroutines to be encapsulated may be entangled with other subroutines developed by many scientists over a long period of time. To solve this problem, we chose to tightly couple the initialization processes of the original GAMESS program and the GAMESS CCA architecture, even though, in the standardized interfaces, it may be possible to use components from other packages for initialization.

In addition, the different parallel mechanisms used in a computation may also hinder the interaction of low-level components in a "plug-and-play" fashion. GAMESS uses the dynamic/static load balance strategies to distribute two-electron integrals across processes, while NWChem and MPQC parallelize the functions that use the two-electron integral calculations. This different design of the parallel mechanisms for 2e-integral calculations will affect the way and the performance of using the integral evaluators from GAMESS and the other two packages. For example, when a GAMESS energy computation uses the 2e-integrals computed by MPQC CCA components, the performance of using integrals from

MPQC may be worse than using integrals from GAMESS since the MPQC integral evaluators can only run sequentially by themselves. Currently, we just limit our application to use the integral evaluators in a single CPU. However, to reach a better or keep the original scalability, chemists from different packages must find out a way to balance the way of using integral evaluators from different packages.

Issues for code efficiency

The integral screening improves the efficiency of integral computations. In GAMESS, screening is a 'built-in' function that is integrated with integral computations and can be turned on or off by setting a flag in the input file. In MPQC, screening is not coupled with integral computations but rather may be performed by the caller of integral computations.

The interfaces for integral and other quantum chemistry computations are defined from a chemistry algorithm point of view. That is, the interfaces for data and methods performing electronic structure calculations are defined, but not for the procedures to improve code efficiency, such as using of screening. On one hand, we want to keep the interfaces as clean as possible, so they should include only data and methods that are essential to a computation; on the other hand, if a technique to improve code efficiency is widely used by every package, we may want to include this technique somewhere in the interface. How to seamlessly integrate via common interfaces computations and their efficient implementations, is a difficult design choice.

Version control and testing procedure

Figure 18 shows the package dependence in this project. Besides three chemistry packages, we also use performance tools provided by TAU [17] to conduct component level performance evaluations. All packages, even compilers, are constantly updated with new versions. Whenever a certain package is updated, all the other packages may require rebuilding, and we have to conduct stability and compatibility testing all over again. The process of rebuilding packages is time consuming; if errors occur during stability and compatibility testing, locating the source of the error is equally time-consuming. When some

bugs are found in a new version of a package, we may have to roll back to an older stable version to continue the development process.

With the scope of quantum chemistry computations and the capabilities provided by the three packages, we expect more components will be developed. Exploring/developing a capable tool to minimize efforts in maintaining/testing packages is essential in a real-size project such as this one.

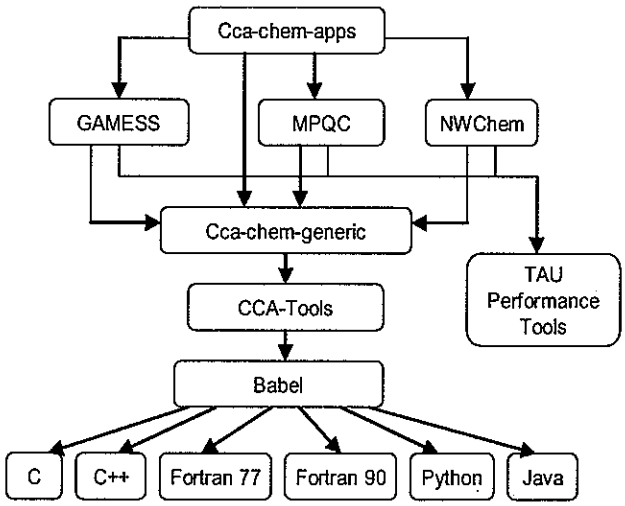

Figure 18. The package dependence for the CCA chemistry project.

Conclusion and future works

In this thesis, I present our experience in developing CCA components based on a large-scale quantum chemistry program. The two parallel mechanisms for GAMESS CCA components and the potential problems for each model are discussed. The process of componentizing GAMESS energy, gradient, Hessian and integral computations is also delineated in detail and issues of interoperability are discussed. This will provide application scientists a perspective about the problems they may be facing when componentizing their packages to explore interoperation with other software. We are extending our experiments to integrate GAMESS and NWChem at the fine-grained level and also build a complete chemistry computation, such as calculating the energy, by using any two of the three chemistry packages through the CCA interfaces. Currently, we have designed the client-side interfaces for integrating GAMESS energy calculation with the other two packages through

integral computations. The implementation of the GAMESS client-side computations is one of our future works.

Based on our experience, community-agreed interfaces and data standards provide only the first step to componentization of a package; substantial efforts are needed to improve the usability of components, control versions of the underlying software, minimize overhead caused by extra layers of function calling, and standardize testing procedures to efficiently explore the errors in coupling many software packages. Componentizing a large-scale legacy software package is an especially challenging task. In other words, comprehensive scientific software engineering is essential in developing components that are truly shareable between scientific packages.

Future works. Integrating GAMESS and NWChem at the fine-grained level, such as on integral calculations, will be one of our future works. We will also build a complete chemistry computation, such as calculating the energy, by using any two of the three chemistry packages through the CCA interfaces.

ACKNOWLEDGEMENTS

I want to give the greatest thank Meng-Shiou Wu from Scalable Computing Laboratory of Ames Laboratory. He is a very generous person and an excellent scientist that always willing to share his knowledge with others. For each paper being published, he is the one that give me the most advises and help me refining the paper.

Thank Ricky A. Kendall from Oak Ridge National Laboratory, who is the one let me starting this project and continuously supporting this research. Thank my co-major professor Masha Sosonkina from Ames Laboratory and the department of computer science. She is wise and active in the CCA community, and always willing to help me and guide me for my research. Thank Mark Gordon from Ames Laboratory and the department of chemistry, who is one of my committee members. As the leader in Ames Laboratory, he tries the best for supporting the researches of his students and is nice to everyone with his warming smiles. Thank my major professor Ying Cai from the department of computer science, who gives me a lot of help in my graduate studies. Without their supports and advises, I would never have been able to start this project and finish what I have done.

Thank Theresa L. Windus from the department of Chemistry and Ames Laboratory in Iowa State University for her help with the chemistry concepts and the NWChem program. Thank Mike Schmidt from the department of Chemistry and Ames Laboratory, Iowa State University for the information on GAMESS and DDI. I also want to thank Joseph P. Kenny from Sandia National Laboratories for his helpful discussions on CCA frameworks and chemistry components.

This work was supported by a SciDAC grant from the Department of Energy via the Ames Laboratory and Sandia National Laboratory. This work was performed at Ames Laboratory under Contract No. DE-AC02-07CH11358 with the U.S. Department of Energy. The United States government has assigned the DOE Report number IS-T 2710.

REFERENCE

[1] Microsoft Corporation, COM, Component Object Model Technologies, 2007, http://www.microsoft.com/com/default.mspx

[2] Steve Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, vol. 14, no. 2, February 1997

[3] Sun Microsystem, Inc., JavaBeans, 2007, http://java.sun.com/products/javabeans/

[4] D. E. Bernholdt, B. A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W.R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, L. F. Diachin, J. A. Kohl, M. Krishnan, G. Kumfert, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. McInnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, and Zhou. S., "A Component Architecture for High-Performance Scientific Computing," *Intl. J. High-Perf. Computing Appl.*, 2004.

[5] CCA, The Common Component Architecture Forum, http://www.cca-forum.org

[6] Babel, January 2004, http://www.llnl.gov/CASC/components/babel.html

[7] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Cordon, J.H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. J. Su, T. L. Windus, M. Dupuis, J. A. Montgomery, "General Atomic and Molecular Electronic Structure System", *J. Comput. Chem.* 14, 1347-1363 (1993)

[8] The Massively Parallel Quantum Chemistry Program (MPQC), Version 2.3.1, Curtis L. Janssen, Ida B. Nielsen, Matt L. Leininger, Edward F. Valeev, Edward T. Seidl, Sandia National Laboratories, Livermore, CA, USA, 2004.

[9] Aprà, E.; Windus, T.L.; Straatsma, T.P.; Bylaska, E.J.; de Jong, W.; Hirata, S.; Valiev, M.; Hackler, M.; Pollack, L.; Kowalski, K.; Harrison, R.; Dupuis, M.; Smith, D.M.A; Nieplocha, J.; Tipparaju V.; Krishnan, M.; Auer, A.A.; Brown, E.; Cisneros, G.; Fann, G.; Fruchtl, H.; Garza, J.; Hirao, K.; Kendall, R.; Nichols, J.; 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.; Jaffé, J.; Johnson, B.; Ju, J.; Kobayashi, R.; Kutteh, R.; Lin, Z.; Littlefield, R.; Long, X.; Meng, B.; Nakajima, T.; Niu, S.; 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 4.7" (2005), Pacific Northwest National Laboratory, Richland, Washington 99352-0999, USA.

[10] J. P. Kenny, S. J. Benson, Y. Alexeev, J. Sarich, C. L. Janssen, L. C. McInnes, M. Krishnan, J. Nieplocha, E. Jurrus, C. Fahlstrom and T. L. Windus, "Component-Based Integration of Chemistry and Optimization Software", *Journal of Computational Chemistry*, 24(14) 1717-1725, 2004

[11] Ryan M. Olson, Michael W. Schmidt, Mark S. Gordon, Alistair P. Rendell, "Enabling the Efficient Use of SMP Clusters: The GAMESS/DDI Model", *SC'03*, November 15-21, 2003, Phoenix, Arizona, USA

[12] MPI, Message Passing Interface Forum, http://www.mpi-forum.org

[13] Boyana Norris, Jaideep Ray, Robert C. Armstrong, Lois C. McInnes, David E. Bernholdt, Wael R. Elwasif, Allen D. Malony, Sameer Shende: Computational Quality of Service for Scientific Components. CBSE 2004: 264-271

[14] Ben Allan, Rob Amstrong, Sophia Lefantzi, Jaideep Ray, Edward Walsh, Pippin Wolfe, Ccaffeine - a CCA component framework for parallel computing, January 2001, http://www.cca-forum.org/ccafe/

[15] Felipe Bertrand and Randall Bramley, "DCA: A distributed CCA framework based on MPI", *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*, April 2004

[16] Gary Kumfert, Scott Kohn, Tammy Dahlgren, Tom Epperly, Steve Smith, and Bill Bosl, "Introducing Babel Decaf.", Common Component Architecture Forum, Indiana University, Bloomington, IN, October 2-3, 2001. LLNL document UCRL-PRES-145982

[17] Madhusudhan Govindaraju, Sriram Krishnan, Kenneth Chiu, Aleksander Slominski, Dennis Gannon, and Randall Bramley, "XCAT 2.0: A Component-Based Programming Model for Grid Web Services", Technical Report-TR562, Department of Computer Science, Indiana University. Jun 2002

[18] Madhusudhan Govindaraju, Michael R. Head, Kenneth Chiu, "XCAT-C++: Design and Performance of a Distributed CCA Framework", The 12th Annual IEEE International Conference on High Performance Computing (HiPC) 2005, pp: 270-279, December 18-21, Goa, India

[19] SCIRun: A Scientific Computing Problem Solving Environment, Scientific Computing and Imaging Institute (SCI), 2007, http://software.sci.utah.edu/scirun.html

[20] V. S. Sunderam, PVM: A Framework for Parallel Distributed Computing, Concurrency: Practice and Experience, 2, 4, pp 315--339, December, 1990

[21] J. Nieplocha, RJ Harrison, and RJ Littlefield, Global Arrays: A nonuniform memory access programming model for high-performance computers, *The Journal of Supercomputing*, 10:197-220, 1996

[22] Attila Szabo and Neil S. Ostlund, Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory, Dover, ISBN 0486691861

[23] Frank Jensen, Introduction to computational chemistry, John Wiley & Sons, ISBN-0471984256

[24] Joseph P. Kenny, Curtis L. Janssen, Edward F. Valeev, and Theresa L. Windus, "Components for Integral Evaluation in Quantum Chemistry", Journal of Computational Chemistry, submitted.

[25] MPQC, The Massively Parallel Quantum Chemistry Program, 2006, http://www.mpqc.org

[26] Tyrrel, Symmetry in Chemistry - Group Theory, January 2000, http://www.science.siu.edu/chemistry/tyrrell/group_theory/sym1.html

[27] GAMESS, The General Atomic and Molecular Electronic Structure System Homepage, 2007, http://www.msg.ameslab.gov/GAMESS/GAMESS.html

[28] Joseph P. Kenny, Common Component Architecture Components for Chemistry, 2007, http://myrmidon.ca.sandia.gov/dokuwiki/doku.php?id=cca_chem:main, October 2007

[29] S. Shende and A. D. Malony, "The TAU Parallel Performance System," (submitted to) *International Journal of High Performance Computing Applications*, ACTS Collection Special Issue, 2005.

[30] Browne, S., Deane, C., Ho, G., Mucci, P. "PAPI: A Portable Interface to Hardware Performance Counters," *Proceedings of Department of Defense HPCMP Users Group Conference*, June 1999.

[31] R. Berrendorf and B. Mohr. "PCL -- The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors". Technical report, Research Centre Juelich GmbH, Juelich, Germany, September 2000.

[32] Tuning and Analysis Utilities, TAU's CCA Tools, http://www.cs.uoregon.edu/research/tau/cca

[33] Advanced Computing Laboratory, Los Alamos National Laboratory: PDT: Program Database Toolkit, Supercomputing '99 flyer, Los Alamos National Laboratory Publication LALP-99-204, November 1999

APPENDIX A. THE GAMESS CLIENT-SIDE INTERFACE

The C++ interfaces for the GAMESS client-side:

```
/**
  * The ClientIntEvalFactory class wraps the
  * IntegralEvaluatorFactory component for different packages.
  */
class ClientIntEvalFactory {

  public:

    /* the reference to an integral evaluator factory */
    Chemistry::QC::GaussianBasis::IntegralEvaluatorFactoryInterface
        evalfactory;

    /*
     * the constructor
     * @package the package that provides integral calculation
     * @molecular the Molecular object stores basis set information
     */
    ClientIntEvalFactory(
        string package,
        Chemistry::QC::QaussianBasis::MolecularInterface molecular)
    {
        // set the package name
        if (package is GAMESS, NWChem or MPQC) package_ = package;
        else package_ = "GAMESS"; // default

        create an evaluator factory "evalfactory",
        this is different for the different package

        // initialize Molecular object
        molecular_ = molecular;
    }

    /* get the package name */
    string get_package() {
        return package_;
    }

    /*
     * get an ClientIntEval1 object with the specified integral type.
     * GAMESS does not provide the integral evaluator1.
     * @type the type of the integral
     */
    ClientIntEval1 get_evaluator1(string type) {
        check to see if the type of integral exists

        // create a composite integral descriptor
        create_descriptor(type);
```

```
    // create an integral evaluator1
    Chemistry::QC::GaussianBasis::IntegralEvaluator1Interface eval1 =
        evalfactory.get_evaluator1(molecular_, descr);

    // create a ClientIntEval1 object
    ClientIntEval1 client_eval1 =
        new ClientIntEval1(type, package_, eval1);

    return client_eval1;
}

/*
 * get a ClientIntEval2 object with the specified integral type.
 * @type the type of the integral
 */
ClientIntEval2 get_evaluator2(string type)
{
    check to see if the type of integral exists

    // create a composite integral descriptor
    create_descriptor(type);

    // create an integral evaluator2
    Chemistry::QC::GaussianBasis::IntegralEvaluator2Interface eval2 =
        evalfactory.get_evaluator2(molecular_, molecular_, descr);

    // create a ClientIntEval1 object
    ClientIntEval1 client_eval2 =
        new ClientIntEval1(type, package_, eval2);

    return client_eval2;

}

/*
 * get a ClientIntEval3 object with the specified integral type.
 * GAMESS does not provide the integral evaluator3.
 * @type the type of the integral
 */
ClientIntEval3 get_evaluator3(string type)
{
        check to see if the type of integral exists

    // create a composite integral descriptor
    create_descriptor(type);

    // create an integral evaluator3
    Chemistry::QC::GaussianBasis::IntegralEvaluator3Interface eval3 =
        evalfactory.get_evaluator3
            (molecular_,molecular_,molecular_,descr);

    // create a ClientIntEval1 object
    ClientIntEval3 client_eval3 =
        new ClientIntEval3(type, package_, eval3);
```

```
        return client_eval3;

    }

    /*
     * get a ClientIntEval4 object with the specified integral type
    * @type the type of the integral
    */
    ClientIntEval4 get_evaluator4(string type)
    {
        check to see if the type of integral exists

        // create a composite integral descriptor
        create_descriptor(type);

        // create an integral evaluator4
        Chemistry::QC::GaussianBasis::IntegralEvaluator4Interface eval4 =
            evalfactory.get_evaluator4
                (molecular_,molecular_,molecular_,molecular_,descr);

        // create a ClientIntEval4 object
        ClientIntEval1 client_eval4 =
            new ClientIntEval1(type, package_, eval4);

        return client_eval4;

    }


  private:

    // the package name
    string package_;

    // the reference to a molecular object
    Chemistry::QC::GaussianBasis::MolecularInterface molecular_;

    /**
      * Create descriptor with the specified integral type.
      * A descriptor is needed for each integral evaluator.
      * @type the integral type
      */
    Chemistry::QC::GaussianBasis::CompositeIntegralDescrInterface
        create_descriptor(string type)
    {
        create a CompositeIntegralDescr object based on the type
        of the integrals
    }

};


/**
  * the implementation of ClientIntEval1 may not be necessary for
  * GAMESS since in GAMESS only 2-center and 4-center integrals are
```

```
  * used.
  */
class ClientIntEval1 {

  public:

    /*
     * constructor
     * initialize an integral evaluator for the specified package and
     * the type of the integral
     */
    ClientIntEval1(
        string type,
        string package,
        Chemistry::QC::GaussianBasis::IntegralEvaluator1Interface eval1)
    {
        type_ = type;
        package_ = package;
        eval1_ = eval1;
    }

    /*
     * get an integral evaluator1
     */
    Chemistry::QC::GaussianBasis::IntegralEvaluator1Interface get_eval1()
    {
        return eval1_;
    }

    /*
     * Set the reference to the integral buffer from the integral
     * evaluator1. This method has to be called before get_array, or any
     * reorder method is called.
     */
    void set_array(
        Chemistry::QC::GaussianBasis::IntegralDescrInterface desc)
    {
        buffer_ = eval1_.get_array(desc);
    }

    /**
     * return the reference to the integral array
     */
    double* get_array()
    {
        return buffer_;
    }

  private:

    // the reference to an integral evaluator1
    Chemistry::QC::GaussianBasis::IntegralEvaluator1Interface eval1_;

    // the type of the integral evaluator1
    string type_;
```

```
    // the package name
    string package_;

    // the one-dimension array that holds the integrals
    // for a one-center integral.
    double* buffer_;
}

class ClientIntEval2 {
  public:

    /**
     * constructor
     * initialize an integral evaluator for the specified package and
     * the type of the integral
     */
    ClientIntEval2(string type, string package,
       Chemistry::QC::GaussianBasis::IntegralEvaluator2Interface eval2)
    {
        type_ = type;
        package_ = package;
        eval2_ = eval2;
    }

    /*
     * get an integral evaluator2
     */
    Chemistry::QC::GaussianBasis::IntegralEvaluator2Interface get_eval2()
    {
        return eval2_;
    }

    /**
     * Set the reference to the integral buffer from the integral
     * evaluator2. This method has to be called before get_array, or any
     * reorder method is called.
     */
    void set_array(
        Chemistry::QC::GaussianBasis::IntegralDescrInterface desc)
    {
        buffer_ = eval2_.get_array(desc);
    }

    /**
     * return the reference to the integral array
     */
    double* get_array()
    {
        return buffer_;
    }

    /**
     * reorder the integrals in the buffer
     * to the integral ordering defined by the cca-chemistry group [24]
```

```
     */
    void reorder_gtom()
    {
        if (package_ == "GAMESS")
            reorder the integrals (in buffer_) to the order used in
            MPQC/NWCHEM
    }

  private:

    /* the reference to an integral evaluator2 */
    Chemistry::QC::GaussianBasis::IntegralEvaluator2Interface eval2_;

    // the type of the integral evaluator2
    string type_;

    // the package name
    string package_;

    // the one-dimension array that holds the integrals
    // for a shell doublet.
    double* buffer_;
};

/**
  * the implementation of ClientIntEval3 may not be necessary for
  * GAMESS since in GAMESS only 2-center and 4-center integrals are
  * used.
  */
class ClientIntEval3 {
  public:

    /**
     * constructor
     * initialize an integral evaluator for the specified package and
     * the type of the integral
     */
    ClientIntEval3(string type, string package,
       Chemistry::QC::GaussianBasis::IntegralEvaluator3Interface eval3)
    {
        type_ = type;
        package_ = package;
        eval3_ = eval3;
    }

    /*
     * get an integral evaluator3
     */
    Chemistry::QC::GaussianBasis::IntegralEvaluator3Interface get_eval3()
    {
        return eval3_;
    }

    /*
     * Set the reference to the integral buffer from the integral
```

```
     * evaluator3. This method has to be called before get_array, or any
     * reorder method is called.
     */
    void set_array(
        Chemistry::QC::GaussianBasis::IntegralDescrInterface desc)
    {
        buffer_ = eval3_.get_array(desc);
    }

    /**
     * return the reference to the integral array
     */
    double* get_array()
    {
        return buffer_;
    }

  private:

    // the reference to an integral evaluator3
    Chemistry::QC::GaussianBasis::IntegralEvaluator3Interface eval3_;

    // the type of the integral evaluator3
    string type_;

    // the package name
    string package_;

    // the one-dimension array that holds the 3-center integrals
    double* buffer_;
};

class ClientIntEval4 {
  public:

    /**
     * constructor
     * initialize an integral evaluator for the specified package and
     * the type of the integral
     */
    ClientIntEval4(string type, string package,
       Chemistry::QC::GaussianBasis::IntegralEvaluator4Interface eval4)
    {
        type_ = type;
        package_ = package;
        eval4_ = eval4;
    }

    /*
     * get an integral evaluator4
     */
    Chemistry::QC::GaussianBasis::IntegralEvaluator4Interface get_eval4()
    {
        return eval4_;
    }
```

```
    /**
     * Set the reference to the integral buffer from the integral
     * evaluator4. This method has to be called before get_array, or any
     * reorder method is called.
     */
    void set_array(
        Chemistry::QC::GaussianBasis::IntegralDescrInterface desc)
    {
        buffer_ = eval4_.get_array(desc);
    }

    /**
     * return the reference to the integral array
     */
    double* get_array()
    {
        return buffer_;
    }

    /**
     * reorder the integrals in the buffer
     * to the integral ordering defined by the cca-chemistry group [24]
     */
    void reorder_gtom()
    {
        if (package_ == "GAMESS")
            reorder the integrals (in buffer_) to the order used in
            MPQC/NWCHEM
    }

  private:

    // the reference to an integral evaluator4
    Chemistry::QC::GaussianBasis::IntegralEvaluator4Interface eval4_;

    // the type of the integral evaluator4
    string type_;

    // the package name
    string package_;

    // the one-dimension array that holds the integrals
    // for a shell quartet.
    double* buffer_;
};



/**
  * to perform GAMESS computation by using integrals
  * from different packages
  */
class GAMESSCCAComputation {
```

```
public:

  /*
   * A ClientIntEvalFactory object for providing
   * integral calculation from GAMESS, MPQC or NWCHEM.
   */
  ClientIntEvalFactory evalfac;

  /* constructor */
  GAMESSCCAComputation()
  {
      integral_package_ = "GAMESS";
      intputfile_ = "";
  }

  /*
   * set the name of the package for integral calculation.
   * the default package is GAMESS.
   */
  void set_integral_package(string integral_package)
  {
      integral_package_ = integral_package;
  }

  /*
   * set the full path to the GAMESS input file
   */
  void set_inputfile(string inputfile)
  {
      inputfile_ = inputfile;
  }

  /*
   * initialize a GAMESS.Model object and
   * a ClientIntEvalFactory object.
   * initialize GAMESS computation.
   */
  int initialize()
  {
      initialize the model object

      // pass the input file for GAMESS wrapper functions to read
      model.setCoordinatesFromFile(inputfile.c_str());

      // initialize the molecular object
      molecular = GAMESS::GaussianBasis_Molecular::_create();
      molecular.initialize("");

      // initialize the ClientIntEvalFactory object
      // cast the GAMESS Molecular object to
      // Chemistry::QC::GaussianBasis::MolecularInterface
      evalfac = new ClientIntEvalFactory(integral_package_,molecular);
  }

  /*
```

```
     * calculate energy by using the integrals provided by
     * the specified package.
     */
    double get_energy() {
        double f = model.get_energy();
        return f;
    }

    /*
     * @type the type of the integrals
     * construct the two-level loop structure to calculate all of
     * shell doublets and use the 1e-integral iteratively
     */
    void compute_oneei(string type)
    {
        // create a ClientIntEval2 object for the specified package
        ClientIntEval2 client_eval2 = evalfac.get_evaluator2(type);

        // pass the reference to a
        // Chemistry::QC::GaussianBasis::IntegralEvaluator2 object
        // to the FORTRAN 77 function
        gamess_eval2(client_eval2.get_eval2());
    }

    /*
     * construct the four-level loop structure to calculate all of
     * shell quartets and use the 2e-integral iteratively.
     */
    void compute_twoei()
    {
        // create a ClientIntEval4 object for the specific package
        ClientIntEval4 client_eval4 = evalfac.get_evaluator4(type);

        // pass the reference to a
        // Chemistry::QC::GaussianBasis::IntegralEvaluator4 object
        // to the FORTRAN 77 function
        gamess_eval4(client_eval4.get_eval4());

    }

private:
    // the name of the package for providing integral calculations
    string integral_package_;

    // the full path to the GAMESS input file
    string inputfile_;

    // the molecular object stores the basis set information
    GAMESS::GaussianBasis_Molecular molecular;

    /*
     * A GAMESS.Model object for initializing GAMESS computation;
     * calculating energy, gradient and Hessian.
     */
    GAMESS::Model model;
```

```
};
```

The interfaces for underlying FORTRAN 77 wrapper functions

```
c     ------------------------------------------------
c     @buffer the integral array that needed to reorder
c     @size the size of the buffer
c     @type the type of the integral (2-center or 4-center)
      subroutine games_reorder(buffer,size,type)
      dimension buffer(size)
      character type*(*)

c     ---------------------------------------------------------------
c     The 2-level loop structure of looping over all shell doublets
c     @eval2 the memory address of an integral evaluator2 object
c     @package the package that provides the integral evaluator2
      subroutine gamess_eval2(eval2, package)
      integer*8 eval2
      character package*(*)

c     we will use the different loop structure for different packages
c     for each iterate, call the compute method of the integral evaluator2
c     to calculate integrals for a shell doublet.

c     if package = MPQC or NWCHEM, gamess_reorder needs to be called
c     before the integrals can be used by GAMESS program

c     ---------------------------------------------------------------
c     The 4-level loop structure of looping over all shell quartets
c     @eval4 the memory address of an integral evaluator4 object
c     @package the package that provides the integral evaluator4

      subroutine gamess_eval4(eval4, package)

      integer*8 eval4
      character package*(*)

c     we will use the different loop structure for different packages
c     for each iterate, call the compute method of the integral evaluator4
c     to calculate integrals for a shell quartet.

c     use the FORTRAN 77 binding of integral evaluator4 for calculating
c     two-electron integrals

c     if package = MPQC or NWCHEM, gamess_reorder needs to be called
c     before the integrals can be used by GAMESS program

c     ---------------------------------------------------------------
c     calculate RHF energy by using the integrals calculated from one
c     of the GAMESS, MPQC and NWChem packages

      subroutine gamess_rhfcl
```

```
c     copy codes from RHFCL subroutine
c     modify the calls to ONEEI to call gamess_eval2
c     modify the calls to TWOEI to call gamess_eval4
```

APPENDIX B. COMMENTS FOR THE COMMON BLOCK "NSHEL"

-ex- Gaussian exponents, for every symmetry unique primitive

-cs- through -ci- are s,p,d,f,g,h,i contraction coefficients normally only one of the -cx- arrays will be non-zero, for any given exponent in -ex-. the exception is "L" shells, where both -cs- and -cp- will have (different) values.

-nshell- is the total number of shells (p shell means x,y,z, d shell means xx,yy,zz,xy,xy,yz, etc.) the various "K"s define each shell's contents:

-katom- tells which atom the shell is centered on, normally more than one shell exists for every atom.

-kloc- gives the location of this shell in the total AO basis, please read the example.

-kstart- is the location of the first exponent and the first contraction coefficient contained in a particular shell. Thus, KLOC is an AO counter, KSTART a primitive counter.

-kng- is the number of Gaussians in this shell, their data are stored consecutively beginning at the -kstart- value.

-ktype- is 1,2,3,4,5,6,7 for s,p,d,f,g,h,i. note that the value stored in -ktype- for an "L" shell is a 2, so that by itself, -ktype- cannot distinguish a "p" from a "L". Thus, KTYPE is one higher than the true angular momentum.

-kmin- and -kmax- are the starting and ending indices of the shell. These are defined as

	s	p	d	f	g	h	i	L
Kmin	1	2	5	11	21	34	57	1
Kmax	1	4	10	20	35	56	84	4

so you can tell an "L" shell by its running from 1 to 4, namely s,x,y,z, whereas a "p" shell runs 2,3,4 for x,y,z. The table above is generated by writing all Cartesian products, "maximum powers first", back to back:

s, x,y,z, xx,yy,zz,xy,xz,yz,
1 2 3 4 5 6 7 8 9 10
xxx,yyy,zzz,xxy,xxz,yyx,yyz,zzx,zzy,xyz, ... g,h,i
11 12 13, 14 15 16 17 18 19 20, ... g,h,i

An example, to try to make this concrete, is a 6-311G(d,p) basis for the molecule CSiH. Just those three atoms, in that order:

	s	L	L	L	d	s	L	L	L	L	d	s	s	s	P
Katom	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3
Kng	6	3	1	1	1	6	6	3	1	1	1	3	1	1	1
Ktype	1	2	2	2	3	1	2	2	2	2	3	1	1	1	2
Kmin	1	1	1	1	5	1	1	1	1	1	5	1	1	1	2
Kmax	1	4	4	4	10	1	4	4	4	4	10	1	1	1	4
Kstart	1	7	10	11	12	13	19	25	28	29	30	31	34	35	36
kloc	1	2	6	10	14	20	21	25	29	33	37	43	44	45	46

-kloc- helps point to the right AO index, e.g. the d shell of the Si atom contains AOs numbered 37,38,39,40,41,42. kloc(i) = kloc(i-1) + kmax(i) - kmin(i) + 1. total number of AOs (NUM in common -infoa-) in this example is 48, from the hypothetical next KLOC of 46 + 4 - 2 + 1.

Clearly -NSHELL- is 15, the number of columns given here.

Note that this example shows you how to tell a -p- from a -L-, even though -ktype- is 2 for each. d shells always have 6 members, for spherical harmonics are not taken care of in the basis (always a Cartesian Gaussian basis is set up) but rather at the time of varying the MOs (either including or omitting the contaminations like xx+yy+zz according to ispher input). If our molecule was really CSiH3, with C3v symmetry, the input gave only one of the hydrogens. The following shows how does -nshel- change by two more atoms,

	s	s	s	p	s	s	s	P
katom	4	4	4	4	5	5	5	5
kng	3	1	1	1	3	1	1	1
ktype	1	1	1	2	1	1	1	2
kmin	1	1	1	2	1	1	1	2
kmax	1	1	1	4	1	1	1	4
kstart	31	34	35	36	31	34	35	36
kloc	49	50	51	52	55	56	57	58

Since these are symmetry equivalent, -kstart- points to the original Gaussian details in -ex- and -cx-, but these are additional AOs, so -kloc- does go up. -nshell- is now 24, and -num- is now 60. a molecule may very well have many hydrogens, perhaps using identical basis sets, but every different set of equivalent hydrogens gets separate storage of its exponents/contraction coefficients (stored at different -kstart- values).

If the molecule has no symmetry (every atom has a new basis set) then the number of primitives is greater or equals the number of atomic orbitals. A basis function, or atomic orbital, those words are the same thing, is a linear combination of at least one Gaussian primitive. When the symmetry of the molecule makes atoms equivalent (in C60, all 60 atoms are the same), GAMESS stores only one such atom's basis. So it is possible, but unlikely, that the number of Gaussians stored in /NSHEL/ could be smaller than the number of AOs.

We don't care very much about the total number of primitives, so the sum of the KNG array is not actually stored! The integral codes loop over NSHELL, picking up the current shell's KATOM, KNG, KMIN and so on. They have an inner loop over the KNG value, and loops from KMIN to KMAX so as to do the integrals over all the primitives. But after the integrals are finished, we only care about how many AOs there are, so NUM in /INFOA/ is saved for the SCF programs to use.