Modernizing the core quantum chemistry algorithms

by

Andrey Asadchev

A dissertation submitted to committee members
in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY

Major: Physical Chemistry

Program of Study Committee:
Mark S. Gordon, Major Professor
Theresa Windus
Brett Bode
Xueyu Song
Phillip Jones

Iowa State University
Ames, Iowa
2012

Table of Contents

Chapter 1. Introduction

The primary goal of computational chemistry is of course to predict chemical properties: energy, gradients, Hessians (vibrational frequencies), and other properties for a given chemical system. For example, to find the excitation energy or rotation barriers one would perform a series of single point energy calculations. To find local extrema on the potential energy surface a series of energy gradient calculations are needed.

The computational science aspect of computational chemistry is often treated as a necessary evil. Over the years, most of the designers and authors of quantum chemistry algorithms and their implementations were chemists and physicists first, and computational scientists second.

The foundations for quantum chemistry were developed before World War II. For example the Hartree-Fock [1,2] method was developed in the late 20's, and the foundation of perturbation theory [3] dates back to the mid 30's. However, practical application of the theoretical methods did not come until the emergence of sufficient computing resources to crunch the numbers.

20th century scientific computing was dominated by Fortran, short for Formula Translator, one of the earliest programming languages, first developed in the 50's [4]. The computers and operating systems at the inception of Fortran were expensive proprietary products, batch machines running stacks of manually prepared inputs. Compared to today's powerful computers, computing in the 50's and the 60's may as well have been done on clay tablets.

In the 70's another language, C [5], and a new operating system, UNIX, came out of Bell Labs. With the rise of UNIX, the C programming language gained strong footing among computer science and computer engineering practitioners. In the same decade Cray produced its first groundbreaking supercomputer, Cray I, which gave researchers for the first time, the ability to crack tough numerical problems, such as weather prediction, in a timely manner. In the field of computational chemistry many of the core programs (some still in use today) were developed and incorporated into computational chemistry

packages, notably HONDO [6] and GAUSSIAN [7]. A majority of this work was spearheaded by John Pople, who won the 1998 Nobel Prize for his contribution to the field.

The 80's saw the growth of UNIX and standardization of system interfaces with POSIX and SystemV [8] standards. C++ [9], a multi-paradigm language based on C, was being developed by Bjarne Stroustrup at Bell Labs. To avoid limitations placed on software by patents and restrictive licensing, Richard Stallman began the GNU foundation, which sought to liberate software development. The GNU Compilers Collection (GCC) and GNU public licenses are perhaps the most visible of the many contributions GNU made to computing and scientific fields. The decade also witnessed the birth of massively parallel supercomputers, such as the Thinking Machines. To take advantage of the emerging trends in scientific computing, a number of parallel computational chemistry algorithms were developed, including parallel Hartree-Fock [10] and second order perturbation theory (MP2) [11]. In the early 80's Purvis and Bartlett first implemented a coupled cluster singles and doubles algorithm [12], or CCSD for short. Subsequently, CCSD with a perturbative triples correction method [13], CCSD(T), was developed which today is the gold standard of computational chemistry. In the same decade, GAMESS [14] began to be developed, with HONDO as much of its initial codebase.

In the 90's, the exotic supercomputers of the previous decades slowly disappeared, starved from the generous military budgets of the Cold War which was now over [15,16]. The burgeoning personal computer market funneled billions of dollars into research and development of commodity Intel and AMD processors. The fragmented UNIX market was slowly eroded by the ever maturing Microsoft Windows and a new operating system, Linux. Started as a hobby in the early 90's by Linus Torvalds, Linux, released under a GNU Public License, quickly caught the interest of programmers worldwide and within a few years became one of the major operating systems of the Internet age. The C++ programming language became the preferred choice for writing complex applications, albeit not just yet in scientific fields. However, more and more scientific codes of the 90's were run and developed for clusters of commodity computers running Linux and connected by relatively inexpensive networks. One of the more interesting developments in computational chemistry was NWChem [17], a set of codes designed specifically with

parallel distributed memory systems in mind. NWChem was perhaps the last major computational chemistry package whose development started primarily in Fortran.

The Internet bubble burst at the turn of the 21st century, spelling financial problems and consequent death to the many flagship companies of the last century, including SUN and SGI. With the release of X86-64 extensions by AMD in 2003, commodity processors became a full-fledged 64-bit architecture, suitable for any computational challenge. By the 2010's the processor market became dominated almost exclusively by multicore AMD and Intel chips, with IBM still retaining some presence in the high-end computing market with its Power processors. The latest development in the commodity computing is the reemergence of accelerators, such as using graphics cards to solve general programs, so-called General Processing on GPU (GPGPU). The leader in the field has been NVIDIA with its CUDA [18] technology, but recently Intel joined the market with its Many Integrated Cores (MIC) technology [18]. The efforts to unify development across regular microprocessors and various accelerators led to OpenCL [18], a set of open standards for developing applications that run across heterogeneous platforms.

The software development in scientific communities has steadily shifted towards C/C++. While there is still a lot of legacy code written in Fortran (and hence continuing development), much of the new development happens in C++ and Python [19]. Examples are Q-Chem [20], with most of its new development happening in C++, and Psi4 [21], almost entirely implemented in C++ with Python used as a scripting engine. The C++ language and compilers continue to evolve and improve at a faster pace than Fortran, mostly due to the influence of the much larger commercial application development market. In terms of raw speed, the C++ programs are as fast as their Fortran counterparts, but C++ has the advantage of modern programming techniques and many libraries and frameworks, e.g. Boost [22].

So, what does the contemporary scientific computing platform look like now? It is almost always a distributed memory cluster of very fast multicore computers, with between 2 and 64 GB of memory per node. Some clusters might have GPU accelerators to augment the computational power. The number of cores in the cluster varies greatly, from just a few to tens of thousands. The interconnect can be 1Gb Ethernet, InfiniBand, of a proprietary

network, such as SeaStar on Cray supercomputers. The file system can be a local disk or a parallel file system capable of storing terabytes of data.

Ultimately, it is the hardware (or rather the hardware limitations) that dictates how the algorithm is to be designed. Until we have infinite memory and bandwidth, the algorithms will always have to be designed with these limitations in mind. Furthermore, the algorithms have to be designed so as to account for a great variety of system configurations. A few general rules of thumb can be used as general guidelines for designing scalable and efficient algorithms: minimize communication, keep memory footprint low and introduce adjustable parameters for memory use, use external libraries, e.g. Linear Algebra Package [23] (LAPACK), and make software easy to modify, extend, and even rewrite, perhaps by using one certain programming language over another. Furthermore, how will the scientific computing landscape look in the future? Who knows! But the software must be designed so that changes dictated by the hardware can be accommodated efficiently.

In the following chapters are attempts to develop a modern, but simple and flexible, C++ foundation for computational chemistry algorithms and several algorithm implementations built upon that foundation with the above rules of thumb in mind.
But before one can get into the intertwined details of science, algorithms, and hardware some theoretical background is necessary to explain to the reader in broad detail the basis sets, two-electron integrals, and transformations which will form the bulk of the subsequent pages.

1. Hartree-Fock

At the center of computational chemistry is the evaluation of the time-independent Schrödinger equation eigenvalue problem,

$$H\Psi = E\Psi$$

where H is the Hamiltonian operator, Ψ is the wavefunction containing all of the relevant information about the chemical system, E is the energy of the system and eigenvalue of the Hamiltonian. To be a proper wavefunction, Ψ must be square integrable and normalized, $<\Psi|\Psi>=1$, and antisymmetric to satisfy the Pauli exclusion requirement for fermions. The expectation value E then can be computed as:

$$<\Psi|H|\Psi>=E$$

In terms of individual contributions, the Schrodinger equation can be written in terms of the kinetic and potential energies of the electrons and nuclei:

$$(T_e + T_n + V_{ee} + V_{en} + V_{nn})\Psi = E\Psi$$

T_e and T_n are the kinetic energy terms for electrons and nuclei respectively

$$T_e = -\sum_{e}^{N_e} \frac{\nabla_e^2}{2}$$

$$T_n = -\sum_{n}^{N_n} \frac{\nabla_n^2}{2m_n}$$

∇^2 is the Laplacian operator,

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$$

V_{ee} is the electron-electron repulsion term,

$$V_{ee} = \sum_{e<f}^{N_e} \frac{1}{r_{ef}}$$

V_{en} is the electron-nucleus attraction term,

$$V_{ee} = -\sum_{e}^{N_e} \sum_{n}^{N_n} \frac{Z_n}{r_{en}}$$

V_{nn} is the nucleus-nucleus repulsion term,

$$V_{nn} = \sum_{n<l}^{N_n} \frac{Z_n Z_l}{r_{nl}}$$

The closed form analytic solution for the Schrodinger equation exists only for the simplest systems, such as those with one or two particles. To evaluate a quantum system

of interest, a number of approximations have to be made. In the Born-Oppenheimer approximation [24] the much slower nuclei are treated as stationary point charges and the Schrodinger equation then reduces to the electronic Schrodinger equation:

$$H_e = T_e + V_{ee} + V_{en}$$

$$H_e \Psi = E_e \Psi$$

The general problem of the type $<\Psi|\frac{1}{r_{ij}}|\Psi>$ has no analytic solution and further approximations must be made. The crudest solution is to assume that electrons do not interact with each other. This leads to the independent particle model in which

$$\Psi_{IPM} = \phi_1(\mathbf{r}_1)\phi_2(\mathbf{r}_2)...$$

is separable with respect to each electron coordinate vector.

The independent particle wavefunction does not satisfy the anti- symmetry requirement, but properties of the determinant do (since exchanging any two rows or columns changes the sign). Taking the determinant of Ψ_{IPM} leads to Slater determinant Ψ_{HF} which in turn leads to the Hartree-Fock method

$$\Psi_{HF}(\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N) = \frac{1}{\sqrt{N!}} \begin{vmatrix} \psi_1(\mathbf{x}_1) & \psi_2(\mathbf{x}_1) & \cdots & \psi_N(\mathbf{x}_1) \\ \psi_1(\mathbf{x}_2) & \psi_2(\mathbf{x}_2) & \cdots & \psi_N(\mathbf{x}_2) \\ \vdots & \vdots & & \vdots \\ \psi_1(\mathbf{x}_N) & \psi_2(\mathbf{x}_N) & \cdots & \psi_N(\mathbf{x}_N) \end{vmatrix}$$

After performing a series of algebraic manipulations, the closed-shell Hartree-Fock energy can be written as

$$E_{HF} = <\Psi_{HF}|H_e|\Psi_{HF}> = 2\sum_i h_{ii} + \sum_{ij}(2J_{ij} - K_{ij})$$

where h_{ii} is the one-electron integral

$$h_{ij} = (\psi_i | h_1 | \psi_j) = \int \psi_i (-\frac{\nabla^2}{2} - \sum_n^{N_n} \frac{Z_n}{r_n}) \psi_j dr_1$$

and the J and K terms, called Coulomb and exchange, respectively, are two-electron integrals

$$J = (ij \mid ij)$$

$$K = (ij \mid ji)$$

$$(ij \mid kl) = \int\int \psi_i(r_1)\psi_j(r_1)\frac{1}{r_{12}}\psi_k(r_2)\psi_l(r_2)dr_1dr_2$$

From now on, the HF label will be dropped and Ψ will be understood to refer to Ψ_{HF} and H to refer H_e.

The only constraint on the one particle orbitals is that they remain orthonormal,

$$< \psi_i \mid \psi_j >= \delta_{ij}$$

Therefore the orbitals can be manipulated to affect the energy. According to the variational principle, the best orbitals are those that minimize the energy,

$$\partial E = \frac{\partial \Psi}{\partial \psi} = 0$$

The method of Lagrange multipliers solves the minimization problem with constraints. The resulting Lagrange equation

$$\partial[< \Psi \mid H \mid \Psi > - \sum_{i,j}(\lambda_{ij} < \psi_i \mid \psi_j > -\delta_{ij})] = 0$$

can be reduced to

$$F\psi_k = \lambda_{ij}\psi_k$$

where F is the Fock operator

$$F = [h_1 + \sum_i (2J_i - K_i)]$$

Taking the Lagrangian multipliers to be of the form

$$\lambda_{ij} = \delta_{ij}\epsilon_k$$

the Hartree-Fock minization problem becomes an eigenvalue problem:

$$F\psi_k = \epsilon_k\psi_k$$

Optimizing general orbitals in the above problem is not generally feasible. Instead Roothaan [25] proposed to expand orbitals in terms of a known basis and restrict the optimization to the coefficients of a known expansion basis $\{\chi\}$:

$$\psi_i = \sum_b^N c_{ib}\chi_b$$

The optimization of molecular orbitals ψ_i in terms of a fixed basis leads to the Hartree-Fock-Roothaan equations

$$FC = \epsilon SC$$

where C is the coefficient matrix and $S = (\chi_i \mid \chi_j)$ is the basis overlap matrix. The above equation is almost a solvable eigenvalue equation, except for the S term. Although a general basis is not usually orthonormal, it can be orthonormalized in which case the overlap matrix becomes the identity matrix, $S = \delta_{ij}$ and the Hartree-Fock-Roothaan equation takes the form of a regular eigenvalue problem

$$FC = \epsilon C$$

Now an expression for the Fock operator can be derived in terms of the coefficients and one- and two- electron integrals over basis functions

$$F = (\chi_i \mid h_1 \mid \chi_j) + D[(\chi_i\chi_j \mid h_2 \mid \chi_k\chi_l) - (\chi_j\chi_k \mid h_2 \mid \chi_j\chi_l)]$$

where $D = 2\sum_i c_{ib} * c_{ib}$ is known as the density matrix.

Since the orbital coefficients appear on both sides of the equation, the Hartree-Fock method must be repeated until the difference between the old and the new coefficients reaches a certain threshold. Because of that, the Hartree-Fock method is also called the self consistent field (SCF) method.

The simple interpretation of the Hartree-Fock method is that an electron is moving in the mean field of the other electrons. The interaction of individual electrons is not correlated, other than accounting for the Pauli exclusion principle. Accounting for electronic interaction will be discussed below.

2. Basis Set

To understand the intricate details of the computational chemistry algorithms, especially when discussing two-electron integrals, a few words must be said about the basis set.

Modern basis sets are based on atomic orbitals, which are spatial orbitals reminiscent of the $s, p, \ldots$ orbital shapes found in physical chemistry books. Because of that the basis sets are often called atomic basis functions or atomic orbitals, as opposed to molecular orbitals, which are simply atomic orbitals transformed via the coefficient matrix C.
The correct shape for an (Cartesian) atomic orbital is the Slater-type orbital (STO)

$$Ax^l y^m z^n e^{-\alpha r}$$

where A is the normalization coefficient and l, m, n are related to the angular momentum quantum number L,

$$L = l + m + n$$

Using a Gaussian function, a similar type of orbital, called Gaussian-type orbital (GTO), can be devised

$$Ax^l y^m z^n e^{-\alpha r^2}$$

Unlike the Gaussian functions, the Slater functions cannot be separated into x, y, z components, making the evaluation of integrals over the Slater basis expensive. On the other hand, Gaussian function can be written as

$$e^{-\alpha r^2} = e^{-\alpha x^2} e^{-\alpha y^2} e^{-\alpha z^2}$$

and due to this property, the computation of integrals over the Gaussian functions is much simpler [26], with a number of different closed-form solutions for one- and two- electron integrals [27,28,29,30]. Most electronic structure programs use GTOs as basis sets. An exception to this trend is Amsterdam Density Functional (ADF) program suite [31] which uses STOs.

To reproduce the approximate shape of an STO, a linear combination of several GTOs can be taken and fitted according to some criteria, a process known as contraction and the resulting orbital called contracted Gaussian-type orbital,

$$\chi_{cgto} = Ax^l y^m z^n \sum_k^K C_k e^{-\alpha_k r^2}$$

where K is the construction order and C_k are the contraction coefficients. In this context, the individual Gaussians are called primitives.

The individual contracted orbitals which share the same primitives are grouped together into shells. The primary reason for doing so is computational efficiency. With a correct algorithm, only the angular term $x^l y^m z^n$ will be different between shell functions; the terms involving expensive exponent computations will be the same.

The simplest contracted basis sets are of the STO-NG family [32], where N is the number of contracted GTOs fitted to an STO using a least-squares method. The major difference between GTOs and STOs is the function shape near the origin, where GTOs are flat and STOs have a cusp. This is especially important for the core electrons near the nucleus. More advanced basis sets typically have more GTOs to represent contracted core orbitals (6-10 GTO) and fewer GTOs to represent non-core orbitals (1-3 GTOs). This segmented approach strikes a delicate balance between accuracy and computational time.

It should be obvious that a larger basis set will give better orbitals and lower energy, based on the Variational Principle. However, larger basis sets will also increase the computational time, may lead to slower convergence, and may result in numerical instabilities. A majority of time is spent evaluating two-electron integrals and building the Fock matrix. Although, atomic integrals do not change from iteration to iteration, storing N^4 elements can be prohibitively expensive for any large system, and thus the integrals can be re-computed on-the-fly. Currently, Hartree-Fock computations with a few thousand basis functions are routinely performed in a matter of hours. In the near future that number is likely to be the tens of thousands.

3. Electron Correlation

As a rule of thumb, the energy computed with the Hartree-Fock method accounts for 99 % of the total electronic energy. However, the desired physical properties are frequently associated with the last 1 % of the energy. Hartree-Fock computations can give very good geometries, but the energy differences can only be qualitative at best.

Recall from the above discussion that the Hartree-Fock model does not account for the instantaneous electronic interaction, but instead treats each electron as interacting with an electronic mean field. The difference between the total energy and the Hartree-Fock energy is called the correlation energy

$$E_{corr} = E_{hf} - E$$

To recover the correlation energy, Hartree-Fock computations must be followed by what are called correlation methods, which try to recover the correlation energy from the Hartree-Fock wavefunction. In the context of electron correlation computations, Hartree-Fock is typically the zeroth order (also called the reference) wavefunction. Among the many correlation methods there are two that are central to the next chapters: the MP2 and coupled cluster methods.

The formula for the MP2 energy is relatively simple, expressed only in terms of integrals over molecular orbitals $(ia \mid jb)$ and orbital energies ϵ

$$E_{MP2} = \sum_{ij} \sum_{ab} \frac{[2(ai \mid bj) - (bi \mid aj)](ai \mid bj)}{\epsilon_i + \epsilon_j - \epsilon_a - \epsilon_b}$$

As is customary in many-body methods, the indices $i, j, ...$ refer to occupied molecular orbitals O, $a, b, ...$ to virtual orbitals V, and p, q, r, s to atomic basis functions N.
The time consuming part of the MP2 energy computation is not the actual energy computation, which scales as O^2N^2, but the transformation from atomic to molecular integrals (also called 4-index transformation), which scale as ON^4. Another bottleneck in many-body methods is the storage of molecular integrals. For a large MP2 calculation the storage may well be on the order of terabytes. The details of the MP2 energy computation will be covered in detail in the corresponding chapter.

Coupled cluster theory was first proposed in nuclear physics [33] and later adopted in quantum chemistry by Cizek [34] as the exponential *ansatz*

$$\Psi = e^T \Psi_0 = e^{(T_1 + T_2 + ... T_n)} \Psi_0$$

where $T_1 ... T_n$ are the n-particle excitation operator and Ψ_0 is the reference wavefunction, typically Ψ_{hf} in computational chemistry. The excitation operator applied to a reference wavefunction is written in terms of excitation amplitudes t from hole states $i, j, k, ...$ (also referred to as occupied orbitals) to particle states $a, b, c, ...$ (also referred to as virtual orbitals),

$$T_n \Psi_0 = \sum_{ijk\ldots} \sum_{abc\ldots} t_{ijk\ldots}^{abc\ldots} \Psi_{ijk\ldots}^{abc\ldots}$$

The CCSD algorithm is an iterative process that scales as $N^2V^2O^2$ and the triples correction (T) scales as N^2V^4O. To compute the CCSD(T) energy, every type of four-index molecular integral is needed. The coupled cluster algorithm will be covered in detail in the last chapter. Both, MP2 and CC can be easily and systematically derived using Goldstone diagrams, a diagrammatic approach to nonrelativistic fermion interaction based on Feynman diagrams. A very thorough treatment of the many-body theory can be found in the excellent book by Shavitt and Bartlet [35].

References

[1] D. R. Hartree. *Proc. Cambridge Phil. Soc.*, 24:89, 1928.

[2] V. Fock. Nherungsmethode zursung des quantenmechanischen mehrkperproblems. *Zeitschrift fr Physik A Hadrons and Nuclei*, 61:126–148, 1930. 10.1007/BF01340294.

[3] C. Moller and M.S. Plesset. Note on an approximation treatment for many-electron systems. *Physical Review*, 46(7):618, 1934.

[4] J.W. Backus, R.J. Beeber, S. Best, R. Goldberg, L.M. Haibt, H.L. Herrick, R.A. Nelson, D. Sayre, P.B. Sheridan, H. Stern, et al. The Fortran automatic coding system. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 188–198. ACM, 1957.

[5] B.W. Kernighan and D.M. Ritchie. *The C programming language*. Prentice Hall, 1988.

[6] Dupuis. M, Rys. J, and King H. F. Hondo. Quantum Chemistry Program Exchange, 11, 336338, 1977.

[7] W. J. Hehre, W. A. Lathan, R. Ditchfield, M. D. Newton, , and J. A. Pople. Gaussian 70. Quantum Chemistry Program Exchange, Program No. 237, 1970.

[8] J. Isaak. Standards-the history of POSIX: a study in the standards process. *Computer*, 23(7):89–92, 1990.

[9] B. Stroustrup. *The C++ programming language*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[10] M. Dupuis and JD Watts. Parallel computation of molecular energy gradients on the loosely coupled array of processors (LCAP). *Theoretical Chemistry Accounts: Theory*,

Computation, and Modeling (Theoretica Chimica Acta), 71(2):91–103, 1987.

[11] J.D. Watts and M. Dupuis. Parallel computation of the Moller–Plesset second-order contribution to the electronic correlation energy. *Journal of computational chemistry*, 9(2):158–170, 1988.

[12] G.D. Purvis III and R.J. Bartlett. A full coupled-cluster singles and doubles model: The inclusion of disconnected triples. *The Journal of Chemical Physics*, 76:1910, 1982.

[13] K. Raghavachari, G.W. Trucks, J.A. Pople, and M. Head-Gordon. A fifth-order perturbation comparison of electron correlation theories. *Chemical Physics Letters*, 157(6):479–483, 1989.

[14] M. S. Gordon and M. W. Schmidt. *Advances in electronic structure theory: GAMESS a decade later*, pages 1167–1189. Elsevier, Amsterdam, 2005.

[15] Cold war's end hits Cray computer. *New York Times*, 1992.

[16] In supercomputers, bigger and faster means trouble. *New York Times*, 1994.

[17] D.E. Bernholdt, E. Apra, H.A. Früchtl, M.F. Guest, R.J. Harrison, R.A. Kendall, R.A. Kutteh, X. Long, J.B. Nicholas, J.A. Nichols, et al. Parallel computational chemistry made easier: The development of NWChem. *International Journal of Quantum Chemistry*, 56(S29):475–483, 1995.

[18] A. Heinecke, M. Klemm, and H. Bungartz. From gpgpu to many-core: Nvidia Fermi and Intel many integrated core architecture. *Computing in Science & Engineering*, 14(2):78–83, 2012.

[19] Python. http://python.org/.

[20] Y. Shao, L.F. Molnar, Y. Jung, J. Kussmann, C. Ochsenfeld, S.T. Brown, A.T.B. Gilbert, L.V. Slipchenko, S.V. Levchenko, D.P. O'Neill, et al. Advances in methods and algorithms in a modern quantum chemistry program package. *Phys. Chem. Chem. Phys.*, 8(27):3172–3191, 2006.

[21] J. M Turney, A. C Simmonett, R. M Parrish, E. G Hohenstein, F. Evangelista, J.T. Fermann, B.J. Mintz, L.A. Burns, J.J. Wilke, M. L Abrams, et al. Psi4: an open-source ab initio electronic structure program. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 2011.

[22] Boost C++ libraries. http://www.boost.org/.

[23] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, et al. Lapack usersguide: Release 1.0. Technical report, Argonne National Lab., IL (United States), 1992.

[24] M. Born and R. Oppenheimer. Zur quantentheorie der molekeln. *Annalen der Physik*, 389(20):457–484, 1927.

[25] C.C.J. Roothaan. New developments in molecular orbital theory. *Reviews of modern physics*, 23(2):69, 1951.

[26] S.F. Boys. Electronic wave functions. i. a general method of calculation for the stationary states of any molecular system. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 200(1063):542–554, 1950.

[27] J.A. Pople and W.J. Hehre. Computation of electron repulsion integrals involving contracted Gaussian basis functions. *Journal of Computational Physics*, 27(2):161–168, 1978.

[28] J. Rys, M. Dupuis, and H. F. King. Computation of electron repulsion integrals using the Rys quadrature method. *Journal of Computational Chemistry*, 4(2):154–157, 1983.

[29] S. Obara and A. Saika. Efficient recursive computation of molecular integrals over Cartesian Gaussian functions. *The Journal of chemical physics*, 84:3963, 1986.

[30] M. Head-Gordon and J.A. Pople. A method for two-electron Gaussian integral and integral derivative evaluation using recurrence relations. *The Journal of chemical physics*, 89:5777, 1988.

[31] G. Te Velde, F.M. Bickelhaupt, E.J. Baerends, C. Fonseca Guerra, S.J.A. van Gisbergen, J.G. Snijders, and T. Ziegler. Chemistry with ADF. *Journal of Computational Chemistry*, 22(9):931–967, 2001.

[32] J.A. Pople. Molecular orbital methods in organic chemistry. *Accounts of Chemical Research*, 3(7):217–223, 1970.

[33] F. Coester and H. Kümmel. Short-range correlations in nuclear wave functions. *Nuclear Physics*, 17:477–485, 1960.

[34] J. Čížek. On the correlation problem in atomic and molecular systems. Calculation of wavefunction components in ursell-type expansion using quantum-field theoretical methods. *The Journal of Chemical Physics*, 45(11):4256–4266, 1966.

[35] I. Shavitt and R.J. Bartlett. *Many-Body Methods in Chemistry and Physics: MBPT and Coupled-Cluster Theory*. Cambridge Molecular Science. Cambridge University Press, 2009.

Chapter 2. New Multithreaded Hybrid CPU/GPU Approach to Hartree-Fock

Andrey Asadchev and Mark S. Gordon

Published in the Journal of Chemical Theory and Computation

Abstract

In this article a new multithreaded Hartree-Fock CPU/GPU method is presented which utilizes automatically generated code and modern C++ techniques to achieve a significant improvement in memory usage and computer time. In particular, the newly implemented Rys Quadrature and Fock Matrix algorithms, implemented as a stand-alone C++ library, with C and Fortran bindings, provides up to 40% improvement over the traditional Fortran Rys Quadrature. The C++ GPU HF code provides approximately a factor of 17.5 improvement over the corresponding C++ CPU code.

1. Introduction

As computer hardware becomes more sophisticated and complex and programming languages, compilers, and software patterns mature, it becomes necessary to re-engineer software written during the eighties or earlier in order to take advantage of modern hardware and language features. Unlike older hardware, modern processors have more and more cores, multithreading becomes more and more important, and novel architectures such as graphical processor units (GPU) enter mainstream scientific computing.

"Legacy" programs often do not take into account low-level details of modern processors such as multilayer cache organization, pipelines, and SIMD (single instruction, multiple data) units[1]. As a result of poor cache performance, programs waste CPU cycles, moving data at the expense of actual computations. Failure to take advantage of the SIMD architecture, due for example to unfavorable control structures and memory access patterns, can lead to as much as a 50 percent drop in performance. Parallel execution within a single node presents a challenge as well: computational tasks in legacy code tend to run as processes, rather than as threads, limiting the utility of shared memory and fast inter-thread communication offered by a multi-threaded environment[2], resulting in

replicated memory which puts additional strain on memory cache and bus. OpenMP can at times solve the problem of multi-threading in legacy codes, provided that internal subroutines are thread-safe, which is not always the case.

There are several projects that aim to address shortcomings of legacy code, implementing the entire suits of Quantum Chemistry algorithms using new programming techniques, typically in C++, for example PSI[3] and MPQC[4].

This paper describes a new approach to the Hartree-Fock method that is meant to address the requirements of modern hardware and software, from a low-level two-electron Rys Quadrature[5] implementation to multi-threaded parallel Fock matrix construction and GPU implementation. The method described here does not aim to replace an entire software package, but rather to provide an independent library that can be used to replace or augment existing Hartree-Fock and integral implementations. This paper is organized as follows: Section II presents the developments associated with the Rys quadrature algorithm, including automatically generated code and the requirements for quartets that contain low and high angular momentum quantum numbers. Section III considers various aspects of the Fock matrix construction. The C++ CPU implementation is presented in Section IV, while the corresponding GPU implementation is discussed in Section V. Section VI considers the performance of the new algorithms, and conclusions are drawn in Section VII.

2. Rys Quadrature Implementation

Modern computers have complex architectures and pipelines, making it difficult for an application programmer to write efficient assembly code. Fortunately, modern compilers are able to produce efficient code if several constraints are met:

- Memory access has a favorable alignment; for example, 16 bytes for the current Intel Core architecture
- Non-overlapping segments of memory are flagged as such, using a special type declaration or compiler pragmas, e.g., the C99 `restrict` keyword
- Innermost loops do not have control statements, such as `if` or equivalent
- Short innermost loops have bounds that are known at compile time

- Innermost memory accesses are contiguous, i.e., they have a stride of one

Provided the above conditions are met, a modern compiler should be able to generate efficient machine code for a particular architecture using advanced features, such as SIMD.

Of course, most application programmers (e.g., computational chemists) would not endeavor to write assembly code. However, nontrivial algorithms, such as the Rys Quadrature that is used for two-electron integrals in quantum chemistry codes[5], still require a significant amount of code to accommodate the compiler requirements. Writing such codes manually can be time consuming and error-prone, regardless of the language used. However, there are a number of code generators which can greatly simplify the task through automation. Using code generators to implement integral routines is not new; for example, the excellent LIBINT[6] library was implemented using a code generator. For this project, the Python Cheetah code generator[7] was chosen for the following reasons:

- Generator statements are embedded directly into the source code template, regardless of language, which, for example, can be C++, C, or Fortran.
- The generator statements are just regular Python statements.
- Any Python module can be imported and used in the generator environment, including several symbolic algebra packages, such as `sympy`[8] and `Sage`[9], which provide an interface with Mathematica[10] and other computer algebra systems.

The strategy towards implementing the Rys Quadrature algorithm is as follows[5b]:

- Certain integrals, particularly those over basis functions with low angular momentum quantum numbers, e.g., L=0 (s) and L=1 (p), and consequently small shell quartet block sizes (e.g., there 64 integrals in a (sp sp|sp s) quartet, and short polynomial expressions, are best computed directly using the entire polynomial expression at once, rather than via two-dimensional intermediates.
- General integrals over basis functions with higher angular momentum quantum numbers have prohibitively long polynomial expressions and must be assembled from two-dimensional intermediate integrals via so called recurrence and transfer relations[5].

2.1. Rys Quadrature

The main idea in the Rys Quadrature is to represent a six dimensional integral

$$(\text{ij}|\text{kl}) = \iint \varphi_i\ 1\ \varphi_j\ 1\ \frac{1}{r_{12}} \varphi_k\ 2\ \varphi_l\ 2\ dr_1\, dr_2$$

as a product of three two-dimensional integrals I_x, I_y. I_z,

$$\sum_a^N I_x\ a\ I_y(a) I_z(a) W(a)$$

summed over an exact N-point numerical quadrature with roots a and weights W. The two-dimensional integrals I_x, I_y. I_z are evaluated using recurrence and transfer equations. The exact formulation of the equations can be found in the original Rys paper[5].

Each primitive integral above corresponds to a single contraction. When evaluating contracted shells, the full expression becomes

$$(ij \mid kl) = \sum_a^A \sum_b^B \sum_c^C \sum_d^D C_a C_b C_c C_d I(a,b,c,d)$$

where the bounds of the summation are shell contraction orders, C are the contraction coefficients and $I(a,b,c,d)$ are primitive uncontracted integrals.

2.2. Small Angular Momentum Integrals

If an integral expression (ij|kl) is simple enough, it can be expanded directly into a polynomial, removing the need to compute and store two-dimensional integrals. Doing this also has the benefit of providing the compiler with enough information to enable aggressive optimization. Furthermore, expanded expressions can be filtered through a computer algebra system, like `Mathematica`, simplified, and organized together arbitrarily. The above strategy is not, however, computationally favorable if the integral expression is large, since the large amount of produced code tends to overflow the data and program cache and can adversely impact performance.

The polynomial expressions are expanded from recurrence and transfer formulas as follows:

- The symbolic algebra Python package, `sympy`, is used to build a raw polynomial expression from terminal terms, the starting and ending values

in the Rys recursive formulas, using recurrence and transfer formulas.

- The raw polynomial expressions are piped into `Mathematica` through `Sage`, a Python package that provides interfaces with popular computer algebra systems. `Mathematica` simplifies the raw polynomial expressions and performs a common sub-expression elimination (CSE) to pull out common terms.
- The number of common terms can be quite large, generally larger than the number of registers (16 for the current generation of Intel x86-64 processors). Simplified expressions are reordered to maximize register reuse.
- Simplified expressions are stored as a plain text Python dictionary dump, together with the terminal terms and common terms expressions.
- Since the expression order may have changed, values might have to be permuted to restore the original integral order

In the expression dictionary dump, each integral block expression has a lookup key, which is a collection of four strings, corresponding to shell symbols. The first entry is the dictionary of terminal symbols (those with empty expressions) and common terms (those with nonempty expressions). The next entry is the list of individual functions in the integral block, specified by their l,m,n angular momentum quantum numbers. Each function has a polynomial expression as a string and a list of required terms, both terminal and common. Once they have been loaded, the expressions can be read from the dictionary and implemented inside the loop over quadrature roots.

The algorithm is fairly straightforward: the primitive integrals, depending on individual contractions of the basis functions i,j,k,l and the corresponding roots and weights a,w of the integral shells P,Q,R,S, are evaluated inside the four nested loops corresponding to primitives. The actual integral construction and summation over the roots is handled by a function specialized for the shell types (e.g., *s, sp, d,* etc.) of the shells P,Q,R,S, i.e. the actual implementation of the polynomial expressions. The bra and ket primitives are pre-computed to reduce the number of exponent computations. Once the integral is assembled for all contractions, it is then reordered to restore the correct order. Finally, the amount of memory required is determined by the integral quartet size. For small integral blocks, this

amount of memory is small enough to completely fit in L1 cache.

Through some experimentation it was found that integral blocks with approximately 160 functions, e.g. $(fsp|sps)$, where *sp* refers to a hybrid *sp* shell, and below tend to have the best balance between performance and code size. Large integral quartets, for example a full SP quartet, tend to increase code size and compilation time dramatically, without noticeable performance benefit.

2.3. General Integrals

General integrals with high angular momentum quantum numbers are best computed using a traditional approach via two-dimensional intermediates. However, the details of the present implementation are significantly different from others and are best described using the pseudo algorithm in the C++ Listing 1. The lines in the pseudo-code after "//" are comments.

```
// P,Q,R,S are the Shell objects that contain all
// information such as contracted Gaussians, angular //
```

```
momentum L, etc.

N = (P+Q+R+S)/2 + 1 // number of quadrature roots
bra (P,Q); // bra primitives

for k,l in (R,S) { // ket contractions
    ket (k,l); // ket primitives
    for i,j in (P,Q) { // bra contractions
        // contraction factor
        C = bra(ij)*ket;
        if (C < cutoff) continue; // screening
        // roots and weights
        (a,w) = roots(bra(ij), ket);
        (Gx,Gy,Gz) = recurrence(bra(ij), ket);
        (Ix(K),Iy(K),Iz(K)) = transfer(Gx,Gy,Gz);
        ++K;
    }
}

for r,s in (R,S) { // R,S functions
    Ix = Ix(:,:,x(r),x(s),:)
    Iy = Iy(:,:,y(r),y(s),:)
    Iz = Iy(:,:,z(r),z(s),:)
    for k in K { // contractions
      for a in N { // roots
        // form integrals
        G(0) += Ix(Li,Lj,k)*Iy(0,0,k)*Iz(0,0,k)
        G(1) += Ix(0,0,k)*Iy(Li,Lj,k)*Iz(0,0,k)
        ...
        G(M-1) += ...
      }
      I(0:M) += C*G
      for a in N { // roots
        ...
      }
      I(M,...) += C*G
      ...
    }
    transform(G)
}
```

Listing 1. Bra Quadrature

The main ideas of the pseudo-code are:

- The bra, $< PQ |$, exponential factors are pre-computed, to avoid a quartic number of exponent computations.
- Inside the individual primitive loops the roots are computed to form recurrence intermediates that in turn are used to generate the final two-dimensional integral via transfer relations for a given contraction K.
- Once all of the two-dimensional integrals are formed. they are transformed

into the final electron repulsion integral (ERI). Details of the implementation are somewhat involved and are explained below.

2.4. Bra Kernel

In calculating the shell functions there is not a simple *runtime* relationship between the number of an iteration, corresponding to a particular basis function $f = (l, m, n)$, and the individual angular momentum quantum numbers l, m, n. Therefore, the angular momentum components could be tabulated and looked up during runtime. However, indirect indexing due to the use of a lookup table prevents effective optimization by the compiler. In the outer loops, there is little overhead due to indexing, but for the innermost loops, corresponding to the bra part, the indexing overhead becomes significant. In order to avoid lookup tables in the bra loops, all of the indexes on the bra side must be available during compilation. This is fairly easy to accomplish using a code generator, the same Python Cheetah code generator described above.

Different kernels, corresponding to different numbers of roots, can also be generated using the code generator. However, since the code described here was written using C++, this becomes unnecessary, since the C++ template meta language can be used to accomplish the same result much more effectively. The number of functions computed in any given block may be too large for the compiler to handle effectively, primarily because there are only a small number of registers. Therefore, the entire list of bra functions is broken up into blocks of M functions each. After some experimentation, an M value of 10 was found to be the most effective.

It should be noted that for a given integral block, the bra subsection is evaluated entirely for each given ket index, for all contractions. This allows the code to generate the entire integral block piecewise, and transform individual bra blocks one by one, without forming the entire integral. The utility of this approach is described in terms of the Fock matrix construction in more detail below.

Throughout the entire computation, the three innermost indices correspond to roots and bra indices that are known at compile time, delegating the task of the actual optimization to the compiler.

3. Fock Matrix Construction Implementation

The construction of the Fock matrix[11] from the integrals and the density matrix can be split into two parts: higher level iterations over the shell quartets and lower-level contraction of the density matrix with the integrals to produce a Fock matrix block that corresponds to a particular integral quartet (i,j,k,l).

The general approach to contracting an integral I with the density matrix D is outlined in Listing 2. The coefficient C refers to Coulomb term coefficients, and X refers to exchange term coefficients. For plain Hartree-Fock (HF) using 8-fold symmetry those coefficients would be 4 and -1 respectively, but for methods that modify the Fock operator, e.g., density functional theory (DFT), those coefficients may be different.

```
// I(i,j,k,l) are already computed: ints
// D(i,j) is density matrix
for l in S { // ket indices
  for k in R {
    for j in Q { // bra indices
      for i in P {
        F(i,j) += C*D(k,l)*I(i,j,k,l)
        F(k,l) += C*D(i,j)*I(i,j,k,l)
        F(i,k) += X*D(j,l)*I(i,j,k,l)
        F(i,l) += X*D(j,k)*I(i,j,k,l)
        F(j,k) += X*D(i,l)*I(i,j,k,l)
        F(j,l) += X*D(i,k)*I(i,j,k,l)
      }
    }
  }
}
```

Listing 2. Fock contraction

The following modifications are made to improve performance:

- The density and Fock matrix blocks, corresponding to a particular combination of two shells, are stored contiguously to optimally use cache locality. This is addressed in more detail in the next subsection.
- The innermost loops are relatively short, and for the best performance the loop sizes are known at compile time.
- The memory usage is dominated by integral storage. However, since the integrals are being formed block by block, the entire integral never needs to be stored. Instead, each bra tile is contracted with the appropriate

density tile to form a Fock tile piece by piece.

- Since small angular momentum integrals are formed at once, a specialized version to handle that case is implemented as well.

The kernel version of the code specialized for the entire bra-ket, i.e. small angular momentum integrals, is essentially Listing 2 with loop bounds that are known at compile time, to provide the compiler with the information needed to enable aggressive optimization. For example, when compiling a Fock kernel corresponding to a (ss|ss) quartet, all the loop bounds are 1 and the compiler will optimize out the loops altogether.

The kernel version specialized for partial Fock contraction is implemented as a function object that "remembers" indices k,l (see the pseudo-code in Listing 3). For each integral bra tile being formed, the apply function is called. With each transformation, the internal indices are updated to maintain the correct state.

```
class Fock {
    k,l = 0 // initial state
    apply(I(P,Q)) {
      for j in Q { // bra indices
        for i in P {
          F(i,j) += C*D(k,l)*I(i,j)
          ...
          F(j,l) += X*D(i,k)*I(i,j)
        }
      }
      ++k // update state indices
      ++l
    }
}
```

Listing 3. Tiled Fock contraction

3.1. Blocking Fock/density matrix

The utility of block partitioning matrix computations is well understood[12]. However, partitioning the Fock matrix into blocks is not straightforward since the block nature of the Fock matrix is determined by the shell order in the basis set. However, the basis set may be sorted in such a way as to group same-size shells together. Reorganizing the basis set alone does not give the Fock matrix a uniform block structure since the basis set typically contains s, p, ... shells. This can be overcome by considering the entire Fock

matrix to be a meta-matrix consisting of sub matrices, each with a uniform block structure, determined by the corresponding shells. Consider a graphical depiction of such a matrix, as shown in Figure 1, showing a hypothetical meta-matrix with a non-uniform block structure organized as uniform matrices. The black lines designate the individual shell block boundaries, with all of the elements inside the block being in a contiguous memory segment. The red graphs show the consecutive layout of blocks in memory, with connected blocks being in the same memory segment in that given order. The blue lines designate the borders of sub matrices, in which all blocks *within* those sub matrices are of uniform dimensions.

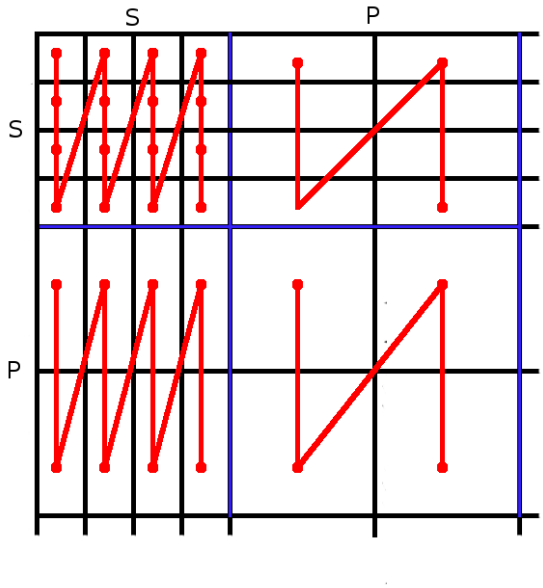

Figure 1. Meta-matrix with block structure

If the programming language constructs allow, the meta-matrix can be given the usual matrix semantics that map individual element access to a specific block in the appropriate sub matrix. In C++ this can be accomplished by defining `operator()(i,j)`. The effect is that a complex meta-matrix can have all three characteristics: sub matrix, block, and element-wise access.

The second benefit of organizing the basis set according to shells is to allow efficient evaluation of multiple similar shell quartets on highly parallel architectures, such as graphical processing units (GPUs). If the shells are grouped together according to coefficients and exponents, as well as the angular momentum quantum numbers, then evaluation of such a block is guaranteed to have the same data except for the Cartesian

centers.

If the Fock matrix needs to be sorted for computational efficiency, the density matrix can be permitted to reflect the desired order. Likewise, if other parts of the program expect the Fock matrix to be in a different order, once formed, the Fock matrix can be un-sorted. This is especially relevant if the Fock matrix is to be used by external programs which may not necessarily sort the basis set.

3.2. Collapsing Fock Algorithm Loops

The regular Fock matrix algorithm, Listing 4, becomes cumbersome if the work has to be divided among different parallel domains and different processors/accelerators. To make the work distribution easier to implement and more efficient, the four nested loops of the Fock algorithm can be collapsed into a single queue-like generator, as illustrated in Listing 5. The basic idea is to map a single index back to four loop indices.

The advantage of using a queue rather than nested loops is that a queue can be transparently and easily parallelized. For the Fock algorithm, the queue tuples are generated on the fly, rather than stored at the expense of N^4 tuples.

The internal counter employed in the queue can be a generic counter, for example, a distributed read-modify-write counter, which allows one to easily transform a seemingly single-node queue into a distributed queue.

```
for l in N {
  for j in N {
    // loop bounds account for 8-fold symmetry
    for k in max(l,j):N {
      for i in j,k+1 {
          ...
      }
    }
  }
}
```

Listing 4. Fock looping

```
class Queue {
    // initial values
    counter = 0;
    last = 0;
    (i,j,k,l) = (0,0,0,0);
    next() {
        next = (i,j,k,l);
        end = (counter++)+1; // advance counter
        for last:end {
            if (empty()) throw exception; // signal if empty
            next = (i,j,k,l);
            i += 1; // i loop
            advance = (i >= (k+1)); // k loop
            if (advance) {
                k += 1;
                i = j;
            }
            advance = advance and (k == N); // j loop
            if (advance) {
                j += 1;
                k = max(j,l);
                i = j;
            }
            advance = advance and (j == N); // l loop
            if (advance) {
                l += 1;
                j = 0;
                k = max(j,l);
                i = j;
            }
        }
        last = end;
        return next;
    }
}
...
while (true) {
    try: (i,j,k,l) = queue.next(); // get next tuple to evaluate
    catch: break; // the end, break from the loop
    ...
}
```

Listing 5. Fock task queue

3.3. Exchanging bra/ket order

Most of the integral algorithms, including the Rys Quadrature, prefer the general integral (pq|rs) over shells P,Q,R,S to be sorted such that $P \geq Q, R \geq S, P \geq R$. Exchanging the order inside the integral code adds complexity and has a performance penalty. But for the

purposes of a Hartree-Fock code, exchanging the order of the quartet indexes alone and of the corresponding sub matrices is sufficient. However, the screening must be done before changing the order if one is using an unmodified screening loop structure.

3.4. Normalization Coefficients

Integrals over functions with angular momentum higher than the P shell must be normalized. The normalization can either be done in the integrals themselves or by absorbing the normalization coefficients into other terms. The advantage of removing normalization coefficients from the integrals is that the integral code is simpler when it is devoid of normalization coefficients.

For the purposes of the HF algorithm, the following approach can be used to shift the normalization coefficients N_i from the integrals to the Fock (F) and density (D) matrices to form normalized matrices F^* and D^*:

$$F_{ij} = (N_i N_j N_k N_l (ij \mid kl)) D_{kl} \tag{1}$$

$$D^*_{kl} = (N_k N_l) D_{kl} \tag{2}$$

$$F^*_{ij} = (ij \mid kl) D^*_{kl} \tag{3}$$

$$F_{ij} = (N_i N_j) F^*_{ij} \tag{4}$$

Therefore, normalization can be handled by first normalizing the density matrix, then performing the regular Fock algorithm, and normalizing the resulting Fock matrix.

3.5. Multithreaded Implementation

A multithreaded Fock algorithm allows one to reduce the memory overhead by maintaining only a single copy of the Fock and density matrices per node. The density matrix, which is read-only, does not need to be protected from conflicting updates. However, the Fock matrix is subject to conflicting simultaneous updates from multiple threads, known as race conditions. For example, evaluating integral quartets (i, j, k, l) with values $(1,1,4,4)$ and $(1,1,3,3)$ requires an update to the Fock elements $F(k,l) = F(1,1)$ in both cases. If the two integral quartets are to be evaluated by two distinct threads, the access to the Fock elements must be synchronized so as to avoid race conditions.

There is a number of ways this can be accomplished. For the best performance an

approach using a matrix block lock/mutex (mutual exclusion object) was chosen. Since the entire Fock matrix can be arbitrarily partitioned into blocks, each block can be given its own mutex that is locked when a thread is ready to update the corresponding block. However it is wasteful to lock the entire Fock matrix block while the integrals are being computed and contracted. A better alternative is for each thread to maintain up to six Fock buffers, $F(i,j)...F(j,l)$, which can then be accumulated into the main shared Fock matrix. The algorithm outline is in Listing 6.

```
for (i,j,k,l) in ERI {
    // thread buffers
    Submatrix f(i,j), ..., f(j,l)
    (f(i,j), ..., f(j,l)) = Contract(Integral(i,j,k,l), D)
    // accumulates submatrix
    for f(m,n) in ((f(i,j), ..., f(j,l))) {
        F.lock(m,n)
        F(m,n) += f(m,n)
        F.unlock(m,n)
    }
}
```

Listing 6. Shared Fock updates

4. C++ Implementation Details

Since the approach detailed in the current work is written in C++, the following libraries and techniques are available:

- Boost libraries[13]
- C++ meta-programming[14], including `boost::enable_if`[15] and `boost::mpl`[16]
- C99 preprocessor and Boost Preprocessor[17]
- OpenMP[18]

The code relies heavily on template meta-programming to accommodate compile time requirements of the integral and Fock kernels and to reduce the amount of boiler-plate copy/paste. Various preprocessor tricks of the Boost Processor are used heavily as well. For example, to "transform" a runtime value into a compile time value, the Boost Preprocessor can be used to generate the transformation, e.g., Listing 7.

BOOST_PP_SEQ_FOR_EACH_PRODUCT will apply a macro ERI for each Cartesian quartet of shell types, automatically creating all possible handlers for a quartet followed by a special case if the quartet is invalid, i.e., not one of the TYPES in the listing below.

```
#define TYPES (SP)(S)(P)(D)(F)//...

void runtime(Quartet quartet) {
    type a = quartet[0];
    type b = quartet[1];
    type c = quartet[2];
    type d = quartet[3];

#define ERI(r, types) \
    if (a == BOOST_PP_SEQ_ELEM(0, types) && \
        b == BOOST_PP_SEQ_ELEM(1, types) && \
        c == BOOST_PP_SEQ_ELEM(2, types) && \
        d == BOOST_PP_SEQ_ELEM(3, types)) { \
        typedef shell_pair<BOOST_PP_SEQ_ELEM(0, types),      \
                           BOOST_PP_SEQ_ELEM(1, types)> bra; \
        typedef shell_pair<BOOST_PP_SEQ_ELEM(2, types),      \
                           BOOST_PP_SEQ_ELEM(3, types)> ket; \
        eri<bra,ket>(quartet);

    BOOST_PP_SEQ_FOR_EACH_PRODUCT(ERI, (TYPES)(TYPES)(TYPES)(TYPES))
    {
        throw invalid_quartet();
    }
}
```

Listing 7. Using preprocessor

The multithreading was implemented using OpenMP. While the Boost Thread library is much more powerful and versatile than OpenMP, only a subset of the multithreading constructs were needed to make the code multithreaded, primarily the loop counter synchronization and mutex constructs. In addition to the above-mentioned libraries, other miscellaneous components from the Boost and Standard Template Library are used throughout.

5. GPU Implementation

There have been various GPU implementations for electron repulsion integrals; for example, the McMurchie-Davidson[19,20], and Rys Quadrature[5b,21,22] approaches. Early on, the GPU implementations primarily targeted single precision computations with s, p functions only, using either CUDA C or accelerator statements. The current generation of

GPU hardware has a much smaller time difference for single vs. double precision, making the case for single precision less obvious.

The authors have utilized double precision exclusively to reproduce the CPU results and to go well beyond s and p functions. The GPU implementation was done using NVIDIA CUDA technology. In developing the GPU implementation of the Hartree Fock method, the following factors are considered:

- High angular momentum and low angular momentum/highly contracted integrals are different in nature and warrant different implementation approaches.
- The integral kernels must be able to evaluate many batches of integrals in one launch. By sorting according to the basis set, a large number of quartets, differing only in the atom centers, but not in shell primitives, can be generated.
- The integrals must be contracted with the density D as soon as possible to reduce the memory overhead from n^4 to n^2 where n is the shell size order, e.g. $n=6$ for a Cartesian d-shell. Therefore, the entire integral quartet is never written into the device memory.
- Contracting integrals with the density directly results in race conditions which must be accounted for.
- Integral batches which cannot be evaluated on the device, must be done on the host.

The current Fermi hardware has 32,768 registers and 48KB of shared memory. The number of concurrent thread blocks is 8. A typical integral kernel will use ~60 registers per thread and 6KB of shared memory. Therefore, up to 8 thread blocks can be executed simultaneously, 64 threads each. The 64 threads are executed in warps, with 32 threads per warp. The threads in each warp are implicitly synchronized but their execution is not implicitly synchronized with the other warp. In essence, a warp can be thought of as an independently executing unit. This fact can be used to partition work along the warp or sub-warp boundaries.

The development of the integral kernels closely follows the CPU version: the implementation is split into general and low angular momentum kernels. The low angular

momentum kernels are parallelized over the contraction loop. In both cases, an efficient implementation requires that the type of integral be known at compile time. This is handled by implementing integrals using C++ templates, with the bra-ket type being a compile time parameter and the shell exponents and coefficients a runtime parameter. The shells, centers, and quartet lists are stored in the device memory. Regardless of the implementation, each kernel loads all three sets of data and forms the corresponding bra-ket primitives in shared memory.

5.1. General Integral Kernel

The general integral kernel is applicable to most combinations of contraction order and bra-ket types. While the general kernel may not perform equally well for some combinations, these combinations can be handled by specialized kernels chosen at runtime.

There are multiple ways one can approach the problem of implementing a general Rys Quadrature algorithm on the GPU architecture. The approach taken here is as follows:

- All roots and weights are computed first and stored in the shared memory first.
- Each thread is assigned a 3-D index corresponding to the recurrence and transfer computations it will perform, where the x index maps to an angular momentum, the y index maps to one of the three Cartesian coordinates , and the z index maps to root.
- The x-index corresponds to either a bra or a ket index. Let L_{ab} be the total bra angular momentum $L_a + L_b$ and N the number of roots. In general, $(L_{ab}+1)*3*N$ threads are needed to evaluate recurrence and $(L_{ab})*3*N$ threads are needed to evaluate transfer, with the higher value being the total number of threads required.

In certain cases, e.g., if one or more of the shells are *S (L=0)*, not all of the recurrence and transfer computations are needed; then, the number of threads will be smaller than $(L_{ab}+1)*3*N$. The computations are independent of one another in the y and z indices, but are dependent on the previous results of a thread with a different x-index (and the same y, z indices). Consider the graphical depiction (Fig. 2) of a transfer relation to form

a *(fd|* bra intermediate from a *(hs|* bra. The y-axis corresponds to the first center of the bra, and the x-axis corresponds to the second center of the bra. Each index (p,q) depends on (p+1,q-1) and (p,q-1). For example, the index (3,2) depends on (3,1) and (4,1) which in turn depend on (3,0), (4,0), (5,0). The intermediate $(4,2)$, computed by thread 4, depends on the value of $(5,1)$ computed by thread 5. To ensure correctness, the work of both threads must be synchronized. If the threads are aligned to 2^n boundaries, such they all fall within the same warp, thesynchronization is implicit. In other words, if the overall number of threads needed is $(L_{ab}+1)*3*N$, padding $(L_{ab}+1)$ to a power of 2 will ensure that all threads with the same y,z indices are in the same warp at the negligible expense of some idle threads.

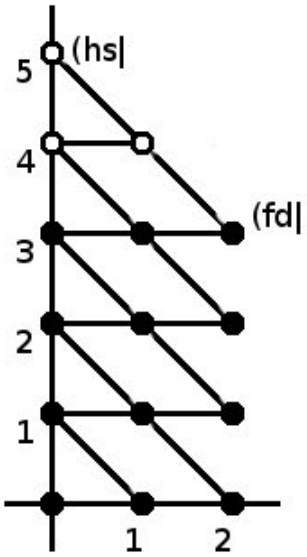

Figure 2. Transfer diagram to form (*fd* | bra

There are three ways the mappings can be aligned to a warp:

(1) The entire recurrence/transfer computation (if small enough) is mapped to a warp (or, a half-warp or a quarter-warp, etc). This holds if $(L_{ab}+1)*3*N \le warp$.

(2) The xy dimension is aligned to a 2^n boundary. For example in the transfer figure above $L_{ab}=5$, the xy-boundary is therefore 16 threads since $L_{ab} * 3 = 15$ and the next power of 2 is $2^n = 16$.

(3) The x dimension is aligned to a 2^n boundary. For example, if $L_{ab}=7$, the x-boundary is 8 threads: since the next power of 2 is $2^n=8$.

Option (1) is preferred. If the first condition fails, the choice between options (2) and (3) depends on which one minimizes the number of threads needed to perform recurrence/transfer computations. For example, if $L_{ab}=4$, recurrence/transfer option (2) needs 16 threads, while option (3) requires 24 threads per root (since the number shown for (3) is per one Cartesian index, it must be multiplied by 3). If $L_{ab}=7$, option (2) needs 32 threads, while option (3) needs 24 threads.

Once the intermediate 2D integrals are in shared memory, each thread computes a subset of integrals. The mapping between a thread/integral index and the corresponding 2D integrals index is stored in the main memory and looked up for each element. The index is stored in a four-element vector, with the fourth index containing the coefficient index for hybrid SP functions.

Once all of the integrals are formed, they are transferred into the shared memory space previously used to store roots and intermediates. The exact number of integrals each thread computes depends on the size of the integral quartets and the number of threads launched. The number of threads depends mostly on the dimensions of the recurrence/transfer computations and the amount of shared memory used by the kernel. To accommodate those two requirements, a number of kernels are available with 2, 3, 4, or 8 multiples of a warp and the corresponding number of integrals per thread. During runtime, the kernel that maximizes the device occupancy is chosen.

For the case in which the entire recurrence/transfer computation can be mapped to a single warp, the integrals can be partitioned to warps rather than to an entire thread block, with each warp assigned to evaluate a unique contraction.

As implemented, the above approach is able to handle any quartet with a total angular momentum of 9 or less, for example $(fd|dd)$, including shells with hybrid sp coefficients. The limit of 9 is imposed by the Rys roots program.

5.2. Low Angular Momentum Integrals

The most natural way to evaluate low angular momentum integrals is to assign individual quartets to a thread block and a single contraction to a thread, with each thread evaluating all integral elements corresponding to that contraction. However, this scheme becomes inefficient if the number of contractions is smaller than the number of threads in a block. This problem can be partially solved by assigning individual roots, rather than contractions, to a thread. For example, for a $(ss|ss)$ quartet this effectively doubles the number of tasks to distribute since for each contraction there are two roots generated.

The low angular momentum kernels reuse the CPU kernel verbatim, with each device thread evaluating an individual root and all of the corresponding integrals, subsequently reduced into shared memory.

Once implemented, the above approach does not saturate the threads. The above implementation was therefore modified to handle an individual quartet per warp, in essence assigning two quartets per thread block. As an additional benefit, shell primitive loads decrease by half.

5.3. GPU Hartree-Fock Implementation

It is not possible to implement a parallel version of the Fock contraction within a thread block in which all six Fock contributions can be evaluated in the single inner loop. The approach taken here is to split the six updates onto separate loops, such that each Fock element can be computed independently. The implementation is as follows:

- One of the six integral/density loops is mapped to a warp. Hence, one thread block can contract and store concurrently one or more Fock tiles corresponding to the integral batch.
- The individual Fock matrix elements are mapped uniquely to a thread in a warp.
- The warp loads the density tile into shared memory.
- The density tile is contracted with the integral batch and the Fock matrix element is stored in a register.
- The Fock matrix is locked with an exclusive read/write lock, and a Fock matrix element is added to the device memory

- The mutex is unlocked and the warp proceeds to contract the next tile.
- Both the density and Fock tiles are stored in a block manner, such that all elements of a tile are continuous in memory.

```
lock(i,j) {
  while (atomicCAS(mutex(i,j),1)) {}
}
unlock(i,j) {
  mutex(i,j) = 0;
}

fock (i, j, k, l) {
  shared G; // integrals
  shared d(k,l); // density tile
  d(k,l) = D(k,l); // load density tile
  f = contract(g,d); // contract
  lock(i,j); // obtain lock
  F(i,j) += f; // add to main memory
  unlock(i,j); // release lock
}

if (do_ij) fock(i, j, k, l);
if (do_kl) fock(k, l, i, j);
if (do_ik) fock(i, k, k, l);
if (do_il) fock(i, l, k, l);
if (do_jk) fock(j, k, i, l);
if (do_jl) fock(j, l, i, k);
```

Listing 8. GPU HF kernel

Only one contraction out of six has a simple indexing; the other five contractions traverse the integrals with a non-contiguous stride, which must be accounted for.

The current CUDA implementation does not provide a built-in device memory mutex, however the mutex can be implemented with the atomic compare and swap operation, `atomicCAS`. The mutex implementation, summarized in Listing 8, will spin until a zero is read. Rather than locking the entire Fock matrix, only the individual tiles are locked at a time.

To achieve performance in the presence of the mutex, the quartets must be traversed so that the indices are not too similar; otherwise one would encounter mutex contention. For example, processing quartets (0,0,0,0),(1,0,0,0),... would result in a high number of collisions as integral quartets are prescreened sequentially. This problem can be avoided

by traversing the quartet list in non-one strides: for example, in strides of 32 in a round-robin manner, provided the quartet lists are on the order of thousands of entries. Since the basis set is sorted to begin with, the generated integral lists are typically well into the thousands.

5.4. Host/GPU Integration

The GPU device is driven by a separate host thread. First, the density matrix is copied into the device memory and the Fock matrix is initialized to zeros. The GPU thread will then request a task from the task queue. If the quartet task can be evaluated by a device kernel, the quartets are prescreened on the host, asynchronously copied to the device and the kernel is launched, asynchronously. This leaves the host thread to either prescreen the next batch or to evaluate those quartets that cannot be handled on the device. This approach allows for the overlap of the CPU/GPU execution. As will be shown in the performance section, the number of unhandled quartets is small, even with a high angular momentum basis set. Once the tasks are exhausted, the Fock matrix on the device is merged into the host.

6. Performance

The newly implemented HF algorithm was compared against the standard GAMESS[23] code, using the Rys Quadrature method only, as well as the default GAMESS option which chooses the optimal integral package according to the integral types[24].

The GAMESS code was compiled with the following command:

```
gfortran -O3 -msse3
```

The new implementation was compiled with:

```
g++ -O3 -msse3
```

The `gcc` version was 4.4.3 for both gfortran and g++. The benchmarks were executed on two Intel Xeon E5405 2.00 GHz CPUs.

The timing comparisons of the new C++ CPU code with the GAMESS code are listed in Table 1. All of the timings are given in seconds, with C++ and GAMESS runs set to utilize a single core. The following should be kept in mind when interpreting the results:

- The rotated axis algorithm and its variations are algorithmically much less complex than the Rys Quadrature algorithm for contracted shells, like

those typically found in low angular momentum basis sets, so GAMESS calculations that use only the Rys algorithm (for comparison purposes) will naturally take longer than the GAMESS default (optimal) option noted above.

- The rotated axis code[24] in GAMESS has been re-implemented to take some advantage of modern processors.
- The Rys Quadrature algorithm is advantageous for small contraction/high angular momentum basis sets. The implementation of the Rys Quadrature algorithm in GAMESS is the original implementation from the HONDO[25] package and does not take into account modern processor architecture.
- For large basis sets with f functions the relative number of shell quartets handled by the Rys Quadrature algorithm is significantly higher than for smaller basis sets.

The test computations were performed on the molecules Cocaine, Taxol, and Valinomycin using basis sets that incorporate a different number of s, p, sp, d, and f shells. Cocaine is the smallest of the three molecules and Valinomycin is the largest. The improvement over the original Rys Quadrature is on the order of 30-40% for all cases. When compared to the default integral option in GAMESS, which picks the Rys Quadrature only if f and higher angular momentum functions are present, the performance is either higher, lower, or the same, depending on the number of d functions, the size of the basis set and correspondingly the memory requirement of the density and Fock matrices.

The rewritten Rys Quadrature algorithm is still much slower than the rotated-shell axis code when only s, p functions are involved. The difference is most pronounced when the total basis set is small. The difference diminishes with increasing Fock and density matrix sizes as memory locality becomes more important. For example, for the Cocaine 6-31G computation, the rotated shell axis code is 75% faster, but only 30% faster with the much larger Valinomycin 6-31G computation.

When d functions are present, the C++ Rys Quadrature code performs better than the current packages as the basis set size increases. For Taxol and Valinomycin, the new CPU

approach outperforms the current GAMESS codes by a few percent. The new code clearly becomes faster if f functions are present. In the best case scenario, it is 31% faster than the GAMESS integral packages, due to both better memory locality and the higher fraction of quartets with higher angular momentum. Overall, the new Hartree-Fock implementation is scalable and efficient, improving the overall performance by as much as 30%.

Table 1. C++ Rys method CPU performance vs GAMESS

System	GAMESS[1]	GAMESS/Rys[2]	C++[3]	Improvement[4] (%)
Cocaine 6-31G	21.3	52.4	37.2	-74.6/29.0 %
Cocaine 6-31G(d)	65.0	112.9	75.2	-15.7/33.4 %
Cocaine 6-31++G(d,p)	402.7	592.0	405.1	-0.60/31.6 %
Cocaine 6-311++G(2df,2p)	3424.4	3686.4	2356.3	31.2/36.1 %
Taxol 6-31G	310.2	691.6	474.1	-52.8/31.4 %
Taxol 6-31G(d)	1104.2	1729.2	1040.0	5.8/39.8 %
Taxol 6-31++G(d,p)	11225.9	15380.5	10288.0	8.4/33.1 %
Valinomycin 6-31G	853.6	1700.7	1104.4	-29.3/35.3 %
Valinomycin 6-31G(d)	2285.0	3445.7	2104.8	7.9/38.9 %

All times are in seconds on a single core

1. GAMESS using various ERI methods (default)
2. GAMESS using only Rys method
3. Newly implemented C++ Rys method
4. Improvement over default GAMESS/ improvement over Rys-only GAMESS

The comparison between the C++ CPU and GPU codes is summarized in Tables 2, 3, 4, broken down by the relative time a particular shell quartet takes. A quartet size is the product of the shell sizes in a quartet. For example, $(ps|ss)$ quartets are of size 3 (3*1*1*1) and $(dd|dd)$ quartets are of size 1296 (6*6*6*6). The benchmark molecule is Taxol and the three basis sets are cc-pVDZ, cc-pVTZ, and 6-31G(d)[26]. The correlation consistent basis sets have contraction orders as high as 4096, while the Pople basis sets rely heavily on hybrid sp shells. Note that a large fraction of integral time is spent computing the multitude of integrals with p shells. In fact, for the cc-pVDZ basis set,

60% of the total time is spent evaluating the smallest (in terms of quartet size) four integrals.

The GPU speed-ups over the single CPU core times (Tables 2,3,4) vary from 17.5x to 12x for the cc-pVTZ basis set. The specialized low-angular momentum quartet kernels perform fairly well, with the lowest speed-up for the last specialized kernel with two sp shells, size 16. The speed-up consequently drops for the general kernel. The performance improves as the quartet gets bigger. The number of slower kernels in the shell size 16-100 range is rather high, and it tends to lower the overall speed-up.

Table 2. Taxol/cc-pVDZ GPU performance

quartet size[1]	CPU % by time[2]	GPU speed-up (x) [3]
1	14.2	35.2
3	22.8	23.0
6	6.6	18.5
9	19.3	17.4
18	9.6	14.5
27	7.0	9.6
36	1.6	11.4
54	8.7	12.6
81	1.9	12.8
108	3.3	17.2
162	2.5	16.0
216	0.4	14.3
324	1.6	16.7
648	0.4	17.9
1296	0.1	15.0
overall [4]	5068.66 s	17.5

[1] product of 4 shell sizes, e.g., s=1, p=3, sp=4, d=6

[2] fraction of total time computing quartet of this size

[3] GPU speed-up (relative to C++ CPU) for quartets of this size

[4] total time and total speed-up

Table 3. Taxol/cc-pVTZ GPU performance

quartet size[1]	CPU % by time[2]	GPU speed-up (x)[3]
1	4.3	25.9
3	8.5	17.5
6	4.6	15.1
9	8.4	13.8
10	1.7	14.6
18	8.0	11.6
27	3.7	8.2
30	3.7	9.3
36	2.5	10.4
54	8.3	11.4
60	2.5	13.3
81	1.1	11.4
90	4.1	15.1
100	0.8	15.5
108	5.8	15.9
162	2.9	15.0
180	5.2	14.0
216	1.2	14.1
270	1.7	17.3
300	1.4	15.8
324	3.5	17.3
360	1.7	15.4
540	3.6	18.9
600	1.1	15.7
648	1.6	17.9
900	1.1	18.7
1000	0.2	15.0
1080	2.9	15.3
1296	0.4	15.8
1800	1.6	19.4
2160	0.7	20.1
3000	0.3	n/a
3600	0.7	n/a
6000	0.3	n/a
10000	0.0	n/a
Overall[4]	35110.4 s	12.0

[1] product of 4 shell sizes, e.g., s=1, p=3, sp=4, d=6

[2] fraction of total time computing quartet of this size

[3] GPU speed-up (relative to C++ CPU) for quartets of this size

[4] total time and total speed-up

Table 4. Taxol/6-31G(d) GPU performance

quartet size[1]	CPU % by time[2]	GPU speed-up (x)[3]
1	1.7	28.5
4	6.5	20.9
6	1.9	18.8
16	12.3	13.1
24	6.6	10.6
36	1.1	11.7
64	13.9	13.7
96	16.8	15.8
144	5.9	19.5
216	0.6	15.4
256	12.4	23.5
384	11.5	20.9
576	7.0	20.2
864	1.7	21.3
1296	0.2	16.6
Overall[4]	1031.94 s	16.6

[1] product of 4 shell sizes, e.g., s=1, p=3, sp=4, d=6

[2] fraction of total time computing quartet of this size

[3] GPU speed-up (relative to C++ CPU) for quartets of this size

[4] total time and total speed-up

CPU and GPU execution can run together to occupy all available resources on the nodes. Table 5 shows the wall clock time required to perform a single SCF iteration of fairly large computations. To showcase various points of performance and comparability, the times are given for combinations of serial and parallel execution with or without GPU.

Table 5. Combined CPU/GPU performance

System	1 core	8 cores	1 GPU	8 cores + 1 GPU
Taxol 6-31G	474.1	60.2	37.4	26.5
Taxol 6-31G(d)	1040.0	132.2	80.2	53.0
Taxol 6-31G(2d,2p)	3429.8	442.3	290.0	178.1
Taxol 6-31++G(d,p)	10288.0	1243.9	984.5	539.9
Valinomycin 6-31G	1104.4	143.9	92.4	60.0
Valinomycin 6-31G(d)	2104.8	270.7	189.6	116.9
Valinomycin 6-31G(2d,2p)	7439.3	964.0	554.0	328.0
All times are in seconds The times include all steps to evaluate a single iteration energy, including diagonalization				

As can be seen, the multithreaded implementation is efficient, consistently achieving over 95% parallel efficiency even for the small computations. Although not shown, the implementation scales well beyond 8 threads. In case of the largest Valinomycin benchmark, combining CPU and GPU execution brought a calculation that took more than two hours to just over 5 minutes.

7. Conclusions

The newly implemented Rys Quadrature and Fock Matrix algorithms, implemented as a stand-alone C++ library, with C and Fortran bindings, provides on the order of 40% improvement over the traditional Fortran Rys Quadrature and performance that is similar to that of less computationally intensive algorithms. The library is fully multithreaded and has favorable scaling across eight cores or more cores within a single node. The library has a simple interface to evaluate a block of integrals as well several compile time parameters to optimize performance. Although algorithmically much more expensive, the new Rys quadrature implementation uses a processor effectively to match and beat the performance of recently implemented algorithms, such as those found in GAMESS[24], which have much less algorithmic complexity for small angular momentum integrals.

The GPU version, adopted from the CPU version, shows speed-ups as high as 17.5x. Importantly, this speedup is relative to the newly optimized C++ CPU code, not to the original legacy Fortran code. The Rys Quadrature however does not scale well in the mid-size shell quartets. Port of a Rotated-Shell axis code is likely to increase the overall performance to 20X or higher.

References

1. Gerber. R. *The software optimization cookbook : high-performance recipes for IA-32 Platforms*. Intel Press, Hillsboro Or., 2006.

2. Gerber. R. *Programming with hyper-threading technology how to write multithreaded software for Intel IA-32 processors*. Intel Press,, Hillsboro, Or., 2004.

3. Turney, J. M., Simmonett, A. C., Parrish, R. M., Hohenstein, E. G., Evangelista, F. A., Fermann, J. T., Mintz, B. J., Burns L. A., Wilke, J. J., Abrams, M. L., Russ, N. J., Leininger, M

. L., Janssen, C. L., Seidl, E. T., Allen, W. D., Schaefer, H. F., King, R. A., Valeev, E. F., Sherrill, C. D., Crawford, T. D., Psi4: *An open source ab initio electronic structure program, Comput*. Mol. Sci., **2011**.

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

5. (a) Rys, J., Dupuis, M., King, H.F., *Computation of electron repulsion integrals using the Rys quadrature method*, J. Comput. Chem., 154–157, *4(2)*, **1983**;

(b) Asadchev, A., Allada, V., Felder, J., Bode, B.M., Windus, T.L., Gordon, M.S., *Uncontracted Rys Quadrature Implementation of up to g Functions on Graphical Processing Units*, J. Comp. Theor. Chem., 696-716, *6*, **2010**.

6. Valeev, E.F., Fermann, J.T., Libint. http://sourceforge.net/p/libint/ (accessed Aug 6, 2012)

7. *Cheetah - the Python-Powered template engine*. http://www.cheetahtemplate.org/ (accessed Aug 6, 2012)

8. *SymPy Development Team. SymPy: Python library for symbolic mathematics*, 2009.

9. Stein, W., Sage: *Open Source Mathematical Software (Version 2.10.2)*. The Sage Group, 2008. http://www.sagemath.org(accessed Aug 6, 2012)

10. Wolfram, S., *The Mathematica(R)book*. Wolfram Media Inc., Champaign IL USA, 5th ed. edition, 2003.

11. Furlani, T.R., King, H.F., *Implementation of a parallel direct SCF algorithm on distributed memory computers*, J. Comput. Chem., 91–104, *16(1)*, **1995**.

12. Buttari, A., Langou, J., Kurzak, J., Dongarra, J., *Parallel tiled QR factorization for multicore architectures*, Proceedings of the 7th international conference on Parallel processing and applied mathematics, PPAM'07

13. *Boost C++ libraries*. http://www.boost.org/ (accessed Aug 6, 2012)

14. Abrahams, D., *C++ template metaprogramming : concepts, tools, and techniques from boost and beyond*, Addison-Wesley, Boston, 2005.

15. *boost::enable_if*. http://www.boost.org/doc/libs/release/libs/utility/enable_if.html (accessed Aug 6, 2012)

16. *The Boost MPL library*
http://www.boost.org/doc/libs/release/libs/mpl/doc/index.html (accessed Aug 6, 2012)

17. *The Boost Preprocessor*
http://www.boost.org/doc/libs/release/libs/preprocessor/doc/index.html (accessed Aug 6, 2012)

18. *Boost thread*. http://www.boost.org/doc/libs/release/doc/html/thread.html (accessed Aug 6, 2012)

19. Ufimtsev, I.S., Martinez, T.J., *Quantum chemistry on graphical processing units. 1. Strategies for two-electron integral evaluation*, J. Chem. Theory Comput, 222–231, *4(2)*, **2008**.

20. Ufimtsev, I.S., Martinez, T.J., *Quantum chemistry on graphical processing units. 2. direct self-consistent-field implementation*. J. Chem. Theory Comput, 1004–1015, *5(4)*, **2009**.

21. Yasuda, K., *Two-electron integral evaluation on the graphics processor unit.* J. Comput. Chem., 334–342, *29(3)*, **2008**.

22. Wilkinson, K.A., Sherwood, P., Guest, M.F., Naidoo K.J., *Acceleration of the GAMESS-UK electronic structure package on graphical processing units*. J. Comput. Chem., 2313–2318, *32(10)*, **2011**.

23. Gordon, M.S., Schmidt, M.W., *Advances in electronic structure theory: GAMESS a decade later*, Theor. Applications Comput. Chem., Ch.. 41, Dykstra, C.E., Frenking, G., Kim, K.S., Scuseria, G.E., Eds., Elsevier, Amsterdam, 2005.

24. Ishimura, K., Nagase, S., *A new algorithm of two-electron repulsion integral calculations: a combination of Pople-Hehre and McMurchie-Davidson methods*. Theor. Chem. Acc,185–189, *120*, **2008**.

25. Dupuis, M., Rys, J., King, H.F., *Hondo*. Quantum Chemistry Program Exchange, 11, 336338, 1977.

26. Davidson, E.R., and Feller, D., *Basis set selection for molecular calculations*. Chem. Rev., 681–696, *86(4)*, **1986**.

Chapter 3. A New Algorithm for Second Order Perturbation Theory

Andrey Asadchev and Mark S. Gordon

Submitted to the Journal of Chemical Theory and Computation

Abstract

A new second order perturbation theory (MP2) algorithm is presented for closed shell energy evaluations. The new algorithm has a significantly lower memory footprint, a lower FLOP (floating point operations) count, and a transparent approach for the disk/distributed memory storage of the MP2 amplitudes. The algorithm works equally well on a single workstation, small cluster, and large Cray cluster. The new algorithm allows one to perform large calculations with thousands of basis functions in a matter of hours on a single workstation. While traditional MP2 calculations are frequently eclipsed by density fitting and resolution of the identity methods, the approaches and lessons learned in the implementation presented here are applicable beyond the MP2 algorithm.

1. Introduction

The integral transformation, also known as the 4-index transformation, is required for many electronic structure computations, including methods that include electron correlation and the analytic computation of energy second derivatives. Of particular interest in the present work is the use of this transformation in second order perturbation theory (called MP2 for second order Moller-Plesset or MBPT2 for second order many body perturbation theory). In general, the 4-index transformation typically transforms atomic integrals to molecular integrals via the simple formula:

$$(ij\,|\,kl) = \sum_p \sum_q \sum_r \sum_s C(i,p)C(j,q)C(k,r)C(l,s)(pq\,|\,rs) \quad (1)$$

Using a common convention, occupied molecular orbitals (o) are designated by indices $i, j, \ldots$, virtual molecular orbitals (v) are designated by indices $a, b, \ldots$, and atomic orbitals (n) are designated by indices p, q, r, s.

Typically, several classes of molecular integrals are needed, e.g., $(ai\,|\,bj)$, $(ab\,|\,ci)$, etc.

In the particular case of MP2, one only needs $(ai|bj)$ integrals to compute the amplitudes and energy, respectively:

$$t_{ij}^{ab} = \frac{2(ai|bj) - (bi|aj)}{\epsilon_i + \epsilon_j - \epsilon_a - \epsilon_b} \tag{2}$$

$$E_{MP2} = \sum_{ab} \sum_{ij} t_{ij}^{ab} (ai|bj) \tag{3}$$

In Eq. (2) the denominator contains the orbital energies for the occupied and virtual molecular orbitals (MOs). Note that only the $(ai|bj)$ integrals (sometimes called (vo|vo) integrals) are needed to form the t_{ij}^{ab} MP2 amplitudes and the MP2 energy. The $(ai|bj)$ integrals, and consequently the t amplitudes have symmetry such that $(ai|bj) = (bj|ai)$ which can be used to halve the storage requirements and the number of computations.

The MP2 energy calculation scales as ON^4 and requires O^2V^2 integral storage, where N, O, and V refer to the number of atomic basis functions, the number of occupied MOs and the number of virtual MOs, respectively. The MP2 method is the least computationally demanding many-body method; it is also the many body method with the lowest compute to I/O ratio.

A number of different MP2 algorithms have been developed over the years[1-6], due to the simplicity of the method and its popularity. The above methods all have advantages and shortcomings. One of the early algorithms is the serial direct method[1]; the integrals are computed on-the-fly and the algorithm does not require any storage other than core memory. However, if the storage required is greater than the available memory, the integrals must be re-evaluated, making the algorithm expensive. The semi-direct serial algorithm[2] avoids integral re-evaluation by storing partially transformed integrals. However, the algorithm does not scale beyond a few hundred basis functions. The parallel direct method[3] scales well as it requires little communication, but it comes at a very high cost of recomputing the integrals. The distributed memory algorithms[4-6] run in parallel, and, using distributed memory to store partially transformed integrals, avoid recomputing the integrals. However, the I/O overhead is high due to poor data locality and the core memory overhead limits the size of the problem, in terms of the numbers of basis functions and occupied orbitals. Furthermore, the algorithm[6], which is implemented

in GAMESS[7], lacks efficiency, because the innermost loops have an unfavorable structure and do not use optimized math routines. The GAMESS disk-based parallel algorithm[8] is a recent improvement over the previously developed algorithms: it has favorable I/O patterns, fast execution, and low memory overhead. Its only drawback is the reliance on fast disk, which is often not available on large clusters with only network file systems.

The aim of the present work is to improve the MP2 algorithm according to the following guidelines:

- Keep the number of operations low and use optimized math libraries to carry out all integral transformations.
- The memory overhead must be low enough to allow computations with several thousand basis functions and several hundred occupied orbitals on current computer hardware. This means that per-core memory overhead must not be more than a gigabyte or two.
- The algorithm must be adaptable to using either a file system or distributed memory as a storage medium. Furthermore, the algorithm should be able to run efficiently on systems with various memory, storage, and interconnect configurations.
- The I/O overhead must be low enough to run off a network file system efficiently

With these guidelines in mind, a new algorithm is developed that runs at least as fast as the current fastest parallel implementation[8], runs equally well on a single workstation and a 1024-core Cray XE6 cluster, can use either disk or distributed memory storage, and can handle an input problem of more than 4000 basis functions.

2. Matrix chaining

There exists a simple matrix multiplication property[9], which, surprisingly, is not very well-known in computational chemistry. Given three (or more) matrices (e.g., B, C, D), the matrices can be multiplied without changing the outcome by two different orders:

$$A = (BC)D \tag{4}$$

$$A = B(CD) \tag{5}$$

At first glance, the above fact may appear to be uninteresting, until one considers the number of operations required for the two expressions. Suppose, for example, that the

general dimensions are $B(k,l)$, $C(l,m)$, $D(m,n)$, and $A(k,n)$. The number of operations are $(klm+kmn)$ and $(lmn+kln)$ for Eqs. (1) and (2), respectively. Of course, if B, C, and D are all square matrices with the same dimension, there is no difference between $(klm+kmn)$ and $(lmn+kln)$.

The difference in the number of operations that are required for Eq. (4) vs. Eq. (5) can be exploited to dramatically reduce the number of operations in integral transformations. Note that the un-factorized $N^4O^2V^2$ complexity of the integral transformation in Eq. (1) can be reduced to either N^4O^1 or N^4V^1 by doing one transformation at a time at the cost of the storage of partially transformed intermediates. Similar to the multiplication schemes described by Eqs. 4 and 5, the integral transformation can be applied in different orders. Suppose, for example, the integral transformation is applied in the naive left-to-right order, virtual index first:

$$(ai\,|\,bj)=\sum_s C(j,s)\sum_r C(b,r)\sum_q C(i,q)\sum_p C(a,p)(pq\,|\,rs) \qquad (6)$$

Then, the total number of operations is:

$$VN^4+VON^3+V^2ON^2+V^2O^2N=VN(N^3+ON^2+VON+VO^2) \quad (7)$$

On the other hand, if the transformation is applied occupied index first,

$$(ij\,|\,ab)=\sum_r C(b,r)\sum_p C(a,p)\sum_s C(j,s)\sum_q C(i,q)(pq\,|\,rs) \qquad (8)$$

then the number of operations is:

$$ON^4+O^2N^3+VO^2N^2+V^2O^2N=ON(N^3+ON^2+VON+V^2O) \quad (9)$$

The expressions in Eqs. (7) and (9) differ by a factor V/O. For correlated calculations, one expects $V>>O$. Therefore, the computational savings obtained by using Eq. (8) rather than Eq. (6) can be significant. The second benefit comes from a reduced memory requirement. Since the first two (inner) transformations contract the first two atomic indices to occupied indices, rather than one virtual/one occupied, the entire tensor is reduced to (o,o,n,n) storage, rather than the much larger (v,o,n,n) storage. For example, with an $(H_2O)_{19}$ water cluster and the aug-cc-pVQZ basis set, the ratio of the two

approaches is 12.3 times with respect to the number of operations and memory.

3. General Algorithm Considerations

To have a scalable algorithm, special attention needs to be paid to the memory footprint, the I/O patterns, and the I/O optimization by means of aggregation of smaller transfers into larger blocks.

3.1. Memory

The algorithm must have a small memory foot print, under 1GB per core on current hardware, even for large computations with several thousand basis functions. In terms of basis functions and shells, the memory overhead must be on the order of M^2O^2, where M is some adjustable blocking factor, for example the size of the largest shell in the basis set. Otherwise, any significant computation would require nodes with ten or more gigabytes of memory per *core*. For example, a computation with 3000 basis functions and 300 occupied orbitals would require 22GB per *core* if memory were to scale as N^2O. The blocking factor must be adjustable to adapt to computers with different number of cores and memory.

3.2. I/O Considerations

For any significant problem size, the sizes of the integral arrays are too great to store in core memory. GAMESS, for example has several MP2 algorithms, two of which are parallel disk-only[8] and distributed memory[6] implementations. However, using modern programming techniques, the same algorithm can be adapted to both disk-based and distributed memory-based approaches. The efficient access patterns between distributed memory and disk are the same: large contiguous transfers are preferred. Typically, a disk-based method has much worse throughput than one that is based on distributed memory. If an algorithm works well with disk, it is guaranteed to work well with distributed memory, even when running over slow Ethernet networks. The general efficacy for using disk has been outlined by Ford, Janowski and Pulay[10]: An important consideration is that individual research groups may not have access to computers with large memory, but access to workstations with large fast disks is very common. There is one important detail: due to buffering, writes tend to be significantly faster than reads. Therefore, algorithms that both read and write large datasets should be optimized in favor of the read operations.

The storage access latency can be hidden by overlapping I/O and computations. This can be accomplished either by having a number of threads perform computations and I/O independently of one another, or by having a single I/O thread perform data transfers while the other thread performs computations.

Implementation transparency; e.g., distributed memory or file implementation, is easily accomplished using polymorphic functions; i.e., function calls that may resolve to two or more implementations during runtime without affecting the logic of the caller. For example, the MP2 program would choose to use distributed memory if enough is available; otherwise it would default to the file system backend. But regardless of the runtime decision, the algorithm itself and its implementation would be exactly the same. In C++, the language of the present implementation, this is done using `virtual` functions.

3.3. File I/O considerations

Two file formats, HDF5[11] and NetCDF[12], and their corresponding libraries allow easy manipulation of multidimensional scientific data on a file system. For the purpose of implementing dense tensor storage, the two file formats are comparable in performance and capabilities.

Storing data on a single node is straightforward. However, parallel storage requires a parallel file system. There are a number of parallel file systems, for example PVFS[13] and Lustre[14]. PVFS is an easily configurable file system, suitable for local clusters. Lustre is a more complicated file system, found for example on large Cray computers. Regardless of a particular file system, the principle is similar to that of RAID0[15] an entire file is striped over a number of I/O nodes. The performance of a parallel file system primarily depends on the stripe size and the number of I/O nodes. Both HDF5 and NetCDF have parallel I/O capabilities.

4. Naive Approach

A simple MP2 approach is described in Listing 1.

```
allocate V(O*O/2,N,N); // (ia|jb) storage
for S in Shells {
  for Q <= S {
    for R in Shells {
      for P in Shells {
        // skip insignificant ints
        if (!screen(P,Q,R,S)) continue;
        t1(i,R,Q,S) = eri(P,Q,R,S)*C(i,P);
      }
      t2(i,j,Q,S) = t1(i,R,Q,S)*C(j,R);
    }
    // exploit symmetry
    V.store(t2(ij,Q,S));
    V.store(t2(ji,S,Q));
  }
}
// 3rd index
for s in N {
  t2(ij,Q) = V(ij,Q,s); // load NO^2 tile
  t3(ij,a) = t2(ij,Q)*C(a,Q); // transform
  V(ij,a,s) = t3(ij,a)); // store VO^2 tile
}
// 4th index + energy computation
for a in V {
  t3(ij,S) = V.load(ij,a,S); // load NO^2 tile
  t4(ij,b) = t3(ij,S)*C(b,S); // transform
  E += Energy(t4); // evaluate energy
}
```

Listing 1. Naive approach

The main points about the simple implementation are:

- The integral symmetry is exploited in the Q, S shells. The half transformed integrals $t2$ are written as triangular matrices, $i \le j$, as well as its transpose $j \le i$. If one is running on multiple cores, each Q, S pair can be evaluated independently, allowing one to benefit from overlapping computation/write.
- The integral computation and first index transformation are screened using the Schwarz method. Subsequent transformations are not screened.
- The matrix transformations can be done using the BLAS matrix routine. Several shells can be transformed at the same time to increase efficiency. The temporary memory is on the order of (O^2M^2).
- The 3rd transformation is straightforward. The required temporary

memory is $(O^2 * N / 2)$ where N is the number of basis functions.

- The fourth transformation requires noncontiguous read. As mentioned above, the disk is not efficient enough to handle noncontiguous read. For a large problem, the 4th step (i.e., the 4th index transformation) becomes increasingly slow, rendering this approach extremely inefficient.

5. Better Algorithm

What is desired is an algorithm that still exploits symmetry and is also able to load integrals with untransformed indexes contiguously to maximize throughput. Suppose the half-transformed integrals $t2$ are stored as a $t2(N*N, ij)$ array, where N is the number of basis functions, and i, j index the already transformed occupied molecular orbitals. Then, it would be a simple matter of reading contiguous blocks corresponding to an occupied index, transforming these blocks, and evaluating the energy, all at the cost of a single read. Note that the quantity $N*N$ is relatively small, only 200MB for 5000 basis functions.

The problem is then how to write such data efficiently since it is generated as a (ij, Q, S) shell pair at the time. Writing individual shell pairs Q, S at a time to form a (QS, ij) set is inefficient. For example, in the case of an s-shell pair, it would require a long noncontiguous write. However, to generate the occupied transformation, very little memory is needed. This fact can be exploited to evaluate and to write a block of M^2 functions at a time. For example, assuming 500 occupied orbitals, the working memory required is 1MB per shell function. Therefore, a block M^2 of 256 functions (e.g. 16 sp-shell pairs) requires only 256MB, but those 16 separate writes can now be aggregated into a single large write. By writing (QS, ij) and its symmetric transpose (SQ, ji) next to each other, the contiguous section of the write can be further doubled.

The fact that the virtual index transformation is also relatively small in terms of memory can be used to further improve the I/O. If an entire *node* has 2 GB of memory, 10 (Q, S, ij) blocks can be loaded at once. This means the tensor storage can be re-dimensioned from $(N*N, O^2/2)$ to $(B*N*N, O^2/(2B))$, with $B=10$ in the example considered here, and consequently the writes can now be $(B*QS, ij/B)$, with atomic and

occupied orbitals interleaved. If $B = O^2 / 2$ then the algorithm can be performed in-core. The graphical depiction of the access patterns is outlined in Figure 1.

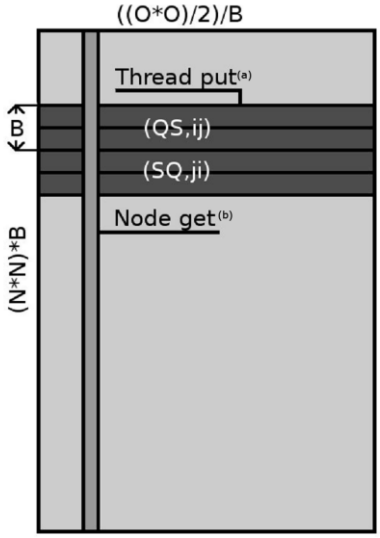

Figure 1. Integral access patterns.

(a) Thread put (shaded) refers to thread I/O to build half-transformed integrals using blocking and symmetry.

(b) Node get (shaded) refers to node-wide I/O to retrieve a contiguous block of half-transformed integrals.

Combining the above ideas, one can develop the following algorithm, Listing 2, which has contiguous writes of size $(2 * M^2 * B)$ and reads of size $(N^2 B)$, with M and B factors determined by setting runtime memory limits.

```
B = ...; // some blocking factor, according to available memory
allocate V( N*N*B, (O^2/2)/B );
// loop over QS pairs in blocks of M functions
for (S,Q) in Blocks(Q <= S, M) {
  for R in Shells.blocks { // loop over R shell blocks
    for P in Shells.blocks { loop over P shell blocks
      // compute ERIs, screening out insignificant integrals
      eri.screen(P,Q,R,S));
      t_(i,S,R,Q) = eri(S,R,Q,P)*C(i,P); // 1st transform
      t1(i,S,Q,R) += t_(i,S,R,Q);
    }
    t2(j,i,S,Q) = t1(i,S,Q,R)*C(j,R); // 2nd transform
  }
  t(QSB,ij/B) = t2(j,i,S,Q); // the shell order is scrambled
  V(QSB,ij/B) = t(QSB,ij/B); // write block
  V(SQB,ji/B) = t(QSB,ij/B); // and symmetrical transpose
}
// 3+4 index transformation and energy
for ij in (O^2/2)/B { // loop over occ. blocks
  t(QSB) = V(QSB,ij); // load scrambled untransformed block
  for (i,j) in B { // loop over occ. indices
    t2(Q,S) = t(QS(i,j)); // unscramble shell order
    t3(a,S) = t2(Q,S)*C(a,Q); // transform
    t4(a,b) = t3(a,S)*C(b,S);
    E += Energy(t4); // compute energy
  }
}
```

Listing 2. Better approach

The main points regarding the above implementation are:

- The integral symmetry is exploited in the Q,S shells. The half-transformed integrals are written independently and can be computed in parallel.
- The Q, S list is processed in terms of blocks of shell pairs, rather than individual shell pairs. The optimal block size will depend on the available memory. The bigger the block size, the better in general.
- The transformed integrals are scrambled such that shells are interleaved with blocks of ij indices of size B. The contiguous size of this noncontiguous write is $2*M^2*B$.

- The transformation of the 3rd and 4th indexes reads the contiguous interleaved blocks. The shell order is unscrambled one occupied pair at a time, the unscrambled block is transformed and the corresponding energy is computed.
- The read operation to fetch the next block can be overlapped with the computations.

The two innermost transformations are responsible for most of the computational work, therefore it is important to have these two as efficient as possible in terms of performance and memory footprint. For any given shell pair (q,s), the entire (P,R) electron repulsion integral (ERI) list is evaluated in terms of blocks of identical shells, to minimize integral initialization overhead. Each individual block is contracted to the first occupied index. Once a given R block is finished, it is then transformed to the second occupied index. Each transformation can be carried out using `dgemm`, making sure that the screened out integrals are absent from the transformation.

6. Performance

A number of benchmarks are useful to judge the performance, scalability, and flexibility of the algorithm:

- How does the new approach compare with similar algorithms?
- How does the network interface affect performance?
- What is the relative time spent in ERI, transformations, and I/O?

The two computer systems used to carry out the benchmarks are:

- Exalted, an Intel cluster connected by InfiniBand (IB). Each node has one 6-core Intel X5650 processor, 24 GB of RAM, one Fermi C2050 graphics processor, and two hard drives in a RAID0 configuration.
- Cray XK6, with Gemini interconnect. Each node has two 16-core AMD 6200 processors, 64 GB of RAM; Lustre parallel file system, 8 I/O nodes.

All inputs used to carry out the benchmarks are listed in Table 1 together with the storage required for the half-transformed integrals. The new implementation is referred to as MP2++ for clarity.

Table 1. Benchmark Specifications

Input	# Basis functions	# Occupied Orbitals	# Virtual Orbitals	Storage Required (GB)
Taxol/6-31G	660	164	434	47
Taxol/6-31G(d)	1032	164	806	115
Taxol/cc-pVDZ	1185	164	959	151
Taxol/aug-cc-pVDZ	2009	164	1659	434
$19H_2\underline{O}$/aug-cc-pVTZ	1995	76	1653	92
Valinomycin/cc-PVTZ	4080	222	3300	3300

First, compare the performance of the new MP2++ CPU-based algorithm to the DDI and IMS implementations in GAMESS on the Exalted cluster connected by InfiniBand. The two inputs are a Taxol molecule, with the small 6-31G and the larger 6-31G(d) basis set, shown in Table 2. Due to the distributed memory (DM) requirement, the DDI algorithm cannot even run on a single node unless the input (i.e., basis set) size is very small. Furthermore, the DDI code is slow compared to both the IMS algorithm and the new implementation, by more than a factor of 10. Furthermore, the DDI MP2 memory requirement scales as ON^2 making it difficult to perform large calculations: A problem larger than 1000 atomic basis functions would require more than 1GB of local memory per *core*, leaving little room to scale.

Table 2. Exalted Benchmarks, compared to DDI[6] and IMS[8].

Input	Cores/Nodes	Algorithm	Storage/Network	Time(mins)*
Taxol/6-31G	24/4	DDI	DM/IB	39.7
		IMS	Disk/IB	3.7
		MP2++	DM/IB	3.1
Taxol/6-31G(d)	36/6	DDI	DM/IB	86.3
		IMS	Disk/IB	7.5
		MP2++	DM/IB	5.4
Taxol/cc-pVDZ	6/1	IMS	Disk/IB	116.6
		MP2++	Disk/IB	76.0
	60/10	IMS	Disk/IB	12.7
		MP2++	DM/IB	$7.9^{(1)}$
			DM/1Gbe	19.3
$(H_2O)_{\underline{19}}$/aug-cc-pVTZ	6/1	IMS	Disk/IB	858.5
		MP2++	Disk/IB	498.3
	60/10	IMS	Disk/IB	121.0
		MP2++	DM/IB	$49.9^{(2)}$
			DM/1Gbe	54.5

* Superscripts $^{(1),(2)}$ refer to computation breakdown, shown in Table 3.

The next set of benchmarks, to illustrate the advantage of the new approach over the IMS algorithm, are Taxol/cc-pVDZ and $(H_2O)_{19}$/aug-cc-pVTZ, given in Table 2. The first example, Taxol/cc-pVDZ, is less computationally intensive but requires 50% more storage (and consequently I/O), whereas the second example, $(H_2O)_{19}$/aug-cc-pVTZ, is computationally heavy due to the presence of diffuse functions. The new implementation is a clear improvement over the existing IMS disk algorithm, especially when diffuse functions are present, being faster by almost a factor of two. The new implementation scales on a small cluster, even when running over the 1Gbe Ethernet interface. The more I/O bound Taxol/cc-pVDZ calculation performance deteriorates quickly. For the computationally heavy water cluster input, the difference between Ethernet and InfiniBand is about 10%.

Table 3. Exalted Parallel Benchmarks Breakdown. All values are the percentage of the total runtime.

Benchmark	ERI	T1	T2	WRITE	READ	T3+T4	sync
(1)	38.2	28.9	6.2	0.77	0.37	23.7	1.9
(2)	39.3	45.5	6.1	0.003	0.7	4.0	4.4

The breakdown of each step in the Taxol/cc-pVDZ and $(H_2O)_{19}$/aug-cc-pVTZ calculations is given in Table 4: T1-T4 are the transformation steps and sync is the overall synchronization time. In both cases the integral calculation (ERI) accounts for a significant fraction of the total run time. The water cluster calculation has almost all of its work concentrated in the integral and first transformation (T1) part due to much less screening (because of the diffuse functions in the basis set), as opposed to the sparser integral set in the Taxol calculation. In both cases, the total I/O (WRITE and READ) accounts for around 1% of the total run time. If the computational power were to suddenly increase, the algorithm would still be viable due to the low I/O overhead.

The next set of benchmarks illustrates the capability of the algorithm on a large cluster, a Cray XE6. Two inputs are used, Taxol/aug-cc-pVDZ and Valinomycin/cc-PVTZ.

The timings are given in Table 4 and the overhead of each step in is given in Table 5.

When considering the timings given below, is important to keep in mind that the numbers are for one thread only and do not give a definitive picture of the entire computer system; the other threads across nodes may very well have significantly more or less I/O time.

Table 4. Cray Benchmarks.

Input	Cores/Nodes	Storage	Time (mins)*
Taxol/aug-cc-pVDZ	512/16	Lustre	63.9(1)
	512/16	DM	52.5(2)
	1024/32	DM	25.3(3)
Valinomycin/cc-PVTZ	256/8	Lustre	313.8(4)
	512/16	Lustre	204.6(5)

* superscripts (1-5) refer to computation breakdown, Table 5.

Table 5. Cray Parallel Benchmarks Breakdown. All values are the percent of total runtime.

Benchmark	Eri	T1	T2	WRITE	READ	T3+T4	sync
(1)	14.1	52.8	10.0	0.1	3.5	5.5	14.0
(2)	17.2	53.1	15.4	0.1	0.1	8.1	6.0
(3)	18.2	40.6	9.9	0.1	0.1	8.4	22.7
(4)	17.8	16.4	18.0	9.3	17.5	18.2	2.8
(5)	7.2	20.5	16.3	18.1	7.1	28.3	2.5

The smaller Taxol/aug-cc-pVDZ computation storage is small enough to fit in distributed memory (DM). The run with Lustre storage takes longer; this can be expected considering that the system has a 64:1 compute to I/O node ratio. When running in distributed memory entirely, the I/O overhead is hardly noticeable, due to the fast Gemini interconnect. The super-linear speed-up is most likely due to the cache effects of reduced memory pressure on individual nodes.

The larger computation, Valinomycin/cc-PVTZ, requires 3.3TB, which is beyond the aggregate memory of the system. The half-integral file is stored on the Lustre file system, with the striping size set to 32MB. For this computation the I/O overhead is significant, on the order of 25%, again due to more effective integral screening in the absence of diffuse functions. The scalability suffers as well, both due to more I/O and an unfavorable 64:1 ratio of cores to I/O nodes when running on 512 cores. Nevertheless, running the calculation that would otherwise require around 2000 cores *just to complete the job* illustrates the efficacy of the new algorithm with its flexible memory/file system storage.

7. GPU Implementation

There is considerable interest in porting core quantum chemistry algorithms to take advantage of the graphical processor unit (GPU) architecture. A previous report by the authors demonstrated reasonable performance for a GPU C++ Hartree-Fock (HF) code, compared to the best C++ CPU code as a benchmark[16]. The speedup of the C++ GPU algorithm relative to the standard FORTRAN77 HF code in GAMESS was shown to be much better, as one would expect when comparing a modest legacy code to a much newer algorithm that takes advantage of modern computer architectures.

With regard to a C++ GPU MP2 code that employs the new algorithm described here, consider the following points regarding the innermost MP2 implementation kernels:

- The integral block evaluated at any given time is relatively small, to keep the memory footprint low.
- The integrals are screened, therefore the coefficient matrix needs to be repacked according to the block-sparse structure of the integral block.
- The first transformation is a series of relatively small matrix-matrix multiplications.

While the CPU can handle the above tasks efficiently, the GPU runtime is inefficient at handling many small tasks, rather than a few large tasks. As a result, the GPU is poorly utilized, even if one uses multiple streams to run several small kernels simultaneously.

Table 6 presents the GPU speed-ups relative to the C++ CPU times discussed above. All benchmarks were carried out on the Exalted nodes. Even though the GPU times are very good relative to the current DDI and IMS algorithms (see Tables 1-3 above), the speed-ups relative the new C++ CPU code is disappointing for the reasons outlined above. The highest performance gain is less than 5%. Although the overall performance of the algorithm is superior to the algorithms in the current GAMESS code, this is primarily due to the new (better) algorithm implementation, rather than to the raw performance of the GPU.

Table 6. GPU. All times are in minutes.

Input	Cores/GPUs	No GPU time	With GPU time	Speed-up[a]

Taxol/6-31G	24/4	3.1	3.4	-8%
Taxol/6-31G(d)	48/8	5.3	5.4	-2%
Taxol/cc-PVDZ	60/10	9.5	9.3	+2%
$(H_2O)_{19}$	60/10	49.9	48.9	+2%

a. Speedup relative to C++ GPU algorithm.

The only place where GPU math libraries could make a difference is in the last two transformations, in which the bulk of the work is handled by two large consecutive matrix multiplies. However the last two index transformations do not account for enough of the runtime, 30% at most in the above examples. Therefore, speeding up those parts of the computations is unlikely to significantly improve the overall performance.

It is important to stress that the above finding does not mean that an efficient MP2 GPU algorithm is not possible. However, to achieve good GPU utilization, an approach significantly different from that taken in the present work is needed. This is in contrast with RI-MP2 GPU implementations[17], in which the bulk of the work is handled by few large matrix multiplies without the need to accommodate the sparse nature of two-electron integrals directly.

8. Conclusions

The work described in this paper offers an improvement over existing MP2 energy algorithms, both in terms of execution time and resource utilization. A flexible data storage model allows one to transparently use either a file system or distributed memory to store partially transformed integrals. A number of sample calculations demonstrate that the new approach works well with small clusters and can also scale to a thousand cores on a Cray supercomputer. However, translating the new C++ CPU approach into a GPU implementation proved to be unsuccessful, since the GPU runtime does not handle the workload composed of large number of small computations efficiently.

Acknowledgements. This work was supported in part by funding from the Nvidia Corporation and the National Science Foundation (Petascale Applications project). The $(H_2O)_{19}$ geometry was provided by Dr. Spencer Pruitt.

References

1. Head-Gordon, M.; Pople, J.A.; Frisch, M.J. MP2 energy evaluation by direct methods. *Chem. Phys. Lett.* **1988**, *153*, 503–506.

2. Frisch, M.J.; Head-Gordon, M.; Pople, J.A. Semi-direct algorithms for the MP2 energy and gradient. *Chem. Phys. Lett*. **1990**, *166*, 281–289.

3. Wong, A.T.; Harrison, R.J.; Rendell, A.P. Parallel direct four-index transformations. *Theor. Chem. Acc.* **1996**, *93*, 317–331.

4. Nielsen, I.; Seidl, E.T. Parallel direct implementations of second-order perturbation theories. *J. Comput. Chem.* **1995**, *16*, 1301–1313.

5. Schütz, M.; Lindh, R. An integral direct, distributed-data, parallel MP2 algorithm. *Theor. Chem. Acc*. **1997**, *95*, 13–34,

6. Fletcher, G.D.; Rendell, A.P.; Sherwood, P. A parallel second-order Møller-Plesset gradient. *Mol. Phys*. **1997**, *91*, 431–438.

7. Gordon, M.S.; Schmidt, M.W.; *Advances in electronic structure theory: GAMESS a decade later, Theory and Applications of Computational Chemistry: the first forty years*; Dykstra, C.E., Frenking, G., Kim, K.S., Scuseria, G.E., Eds., Elsevier, Amsterdam, 2005, 1167-1189

8. Ishimura, K.; Pulay, P.; Nagase, S. A new parallel algorithm of MP2 energy calculations. *J. Comput. Chem.* **2006**, *27*, 407–413.

9. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. *Introduction to Algorithms*; MIT Press and McGraw-Hill, 2001, MA, 331-338.

10. Ford, A.R.; Janowski, T.; Pulay, P. Array files for computational chemistry: MP2 energies. *J. Comput. Chem*. **2007,** *28*, 1215–1220.

11. The HDF Group. Hierarchical Data Format, version 5. http://www.hdfgroup.org/HDF5 (accessed Oct 6, 2012)

12. Network Common Data Form. http://www.unidata.ucar.edu/software/netcdf/ (accessed Oct 6, 2012)

13. Lustre. http://lustre.org/ (accessed Oct 6, 2012)

14. Parallel Virtual File System, version 2. http://pvfs.org/ (accessed Oct 6, 2012)

15. Patterson, D.A.; Gibson, G.; Katz, R.H. A case for redundant arrays of inexpensive disks (RAID). *SIGMOD Rec.* **1988,** *17*, 109–116,

16. Asadchev A.; Gordon, M. S. New Multithreaded Hybrid CPU/GPU Approach to Hartree-Fock. *J. Comp. Theor. Chem.* accepted.

17. Watson, M.; Olivares-Amaya, R.; Edgar, R.G.; Aspuru-Guzik, A. Accelerating correlated quantum chemistry calculations using graphical processing units. *Comput. Sci. Eng*. **2010,** *12*, 40–51.

Chapter 4. A Novel Approach to CCSD(T)

Andrey Asadchev and Mark S. Gordon

Department of Chemistry

Iowa State University and Ames Laboratory

Ames, IA 50011

Abstract

A new coupled cluster singles and doubles with triples correction, CCSD(T), algorithm is presented. The new algorithm is implemented in C++, has a low memory footprint, fast execution time, low I/O overhead, and a flexible storage backend with the ability to use either distributed memory or a file system for storage. The algorithm is demonstrated to work well on single workstations, a small cluster, and a high-end Cray computer. With the new implementation, a CCSD(T) calculation with several hundred basis functions and a few dozen occupied orbitals can run in under a day on a single workstation.

1. Introduction

As a rule of thumb, the electronic energy obtained with the Hartree-Fock method accounts for ~99 % of the energy. However, many chemical properties of interest are dependent on the remaining 1 %, frequently called the electron correlation energy, or simply the correlation energy. The correlation energy is defined as the difference between the reference Hartree-Fock energy and the true energy,

$$E_{corr} = E_{hf} - E \qquad (1)$$

Of the many electron correlation methods [1-3], the coupled cluster (CC) method is one of the most successful. The coupled cluster method was first developed by nuclear physicists [4], adapted to quantum chemistry by Cizek, Paldus, Shavitt, Mukherjee, Schaefer, and others [5-9] and especially popularized by Bartlett [10].

The iterative singles and doubles coupled cluster (CCSD), plus triples that are included perturbatively [11], CCSD(T), method is the most popular approach, often referred to as the gold standard of computational chemistry, among the several other higher-order methods [12].

The coupled cluster method is usually introduced in the exponential *ansatz* form **[**13**]**,

$$\Psi = e^{T}\Psi_0 = e^{(T_1+T_2+\ldots T_n)}\Psi_0 \qquad (2)$$

where $T_1 \ldots T_n$ are the n-particle cluster operators and Ψ_0 is the reference wavefunction,

typically the Hartree-Fock reference Ψ_{hf}.

The excitation operator applied to a reference wavefunction is written in terms of cluster excitation amplitudes t from hole states $i,j,k,...$ (occupied orbitals in chemistry parlance) to particle states (or virtual orbitals), $a,b,c,...$

$$T_n\Psi_0 = \sum_{ijk...}\sum_{abc...} t_{ijk...}^{abc...}\Psi_{ijk...}^{abc...} \qquad (3)$$

Truncating the expansion at doubles, leads to the approximate coupled cluster singles and doubles method, CCSD,

$$T \approx T_1 + T_2 \qquad (4)$$

The singles t_i^a and doubles t_{ij}^{ab} amplitudes are found by solving a system of nonlinear equations,

$$< \Phi_i^a \mid (H_N e^{T_1+T_2}) \mid \Phi >= 0 \qquad (5)$$

$$< \Phi_{ij}^{ab} \mid (H_N e^{T_1+T_2}) \mid \Phi >= 0 \qquad (6)$$

where $\Phi, \Phi_i^a, \Phi_{ij}^{ab}$ are, respectively, the reference determinant, and the singly and doubly excited determinants, and $H_N = H - < \Phi \mid H \mid \Phi >$ is the normal order Hamiltonian [13], constructed so that its reference energy is zero.

The final algebraic CC equations, derived using a diagrammatic approach, result in a number of integral terms V contracted with T amplitudes. For example, VT_1^2 signifies integral terms contracted with $t_i^a t_j^b$. The complete derivation can be found in a number of sources [14]. For the purposes of this work, the spin-free equations by Piecuch and co-workers [15] are used.

The algebraic CC equations are presented in Einstein summation terminology, in which repeated co- and contra-variant indices; e.g., the index s in $X_s^r Y_t^s$ or the index r in $X_s^r Y_r^t$ imply summation. For the following discussion, define the one-electron integrals $f_q^p = < p \mid f \mid q >$, the two-electron molecular integrals $v_{rs}^{pq} = < pq \mid v \mid rs >$, and the many-body denominators $D_{qs...}^{pr...} = f_q^q + f_s^s + ... - f_p^p - f_r^r - ...$ for an arbitrary number of

orbitals. Now, the CCSD non-linear equations may be expressed as follows:

$$D_i^a t_i^a = f_i^a + I_e^a t_i^e - I'^m_i t_m^a + I_e^m(2t_{mi}^{ea} - t_{im}^{ea}) + t_m^e(2v_{ei}^{ma} - v_{ei}^{am}) - v_{ei}^{mn}(2t_{mn}^{ea} - t_{mn}^{ae}) + v_{ef}^{ma}(2t_{mi}^{ef} - t_{im}^{ef}) \tag{7}$$

$$D_{ij}^{ab} t_{ij}^{ab} = v_{ij}^{ab} + P(ia/jb)[t_{ij}^{ae} I_e^b - t_{im}^{ab} I_j^m + \frac{1}{2} v_{ef}^{ab} c_{ij}^{ef} + \frac{1}{2} c_{mn}^{ab} I_{ij}^{mn} - t_{mj}^{ae} I_{ie}^{mb} - I_{ie}^{ma} t_{mj}^{eb} + (2t_{mi}^{ea} - t_{im}^{ea}) I_{ej}^{mb} + t_i^e I'^{ab}_{ej} - t_m^a I'^{mb}_{ij}] \tag{8}$$

In Eqs. (7) and (8), the intermediates c, I, I' are defined as

$$I_a^i = f_a^i + 2v_{ae}^{im} t_m^e - v_{ea}^{im} t_m^e \tag{9}$$

$$I_b^a = (1 - \lambda_a^b) f_b^a + (2v_{be}^{am} t_m^e - v_{be}^{ma} t_m^e) - (2v_{eb}^{mn} c_{mn}^{ea} - v_{be}^{mn} c_{mn}^{ea}) - t_m^a f_b^m \tag{10}$$

$$I_j^i = I'^i_j + I_e^i t_j^e \tag{11}$$

$$I'^i_j = (1 - \lambda_j^i) f_j^i + (2v_{je}^{im} t_m^e - v_{ej}^{im} t_m^e) + (2v_{ef}^{mi} t_{mj}^{ef} - v_{ef}^{im} t_{mj}^{ef}) \tag{12}$$

$$I_{kl}^{ij} = v_{kl}^{ij} + v_{ef}^{ij} c_{kl}^{ef} + P(ik/jl) t_k^e v_{el}^{ij} \tag{13}$$

$$I_{jb}^{ia} = v_{jb}^{ia} - \frac{1}{2} v_{eb}^{im} c_{jm}^{ea} - v_{jb}^{im} t_m^a + v_{eb}^{ia} t_j^e \tag{14}$$

$$I'^{ab}_{ci} = v_{ci}^{ab} - v_{ci}^{am} t_m^b - t_m^a v_{ci}^{mb} \tag{15}$$

$$I'^{ia}_{jk} = v_{jk}^{ia} + v_{ef}^{ia} t_{jk}^{ef} + t_j^e t_k^f v_{ef}^{ia} \tag{16}$$

$$c_{ab}^{ij} = t_a^i t_b^j + t_{ab}^{ij}$$

In the foregoing, the permutation operator P,

$$P(ia/jb) u_{ab}^{ij} = u_{ab}^{ij} + u_{ba}^{ji} \tag{17}$$

has the effect of symmetrizing an arbitrary operand u, such that,

$$P(ia/jb) u_{ab}^{ij} = P(ia/jb) u_{ba}^{ji} \tag{18}$$

The integrals over molecular orbitals are obtained from the integrals over the atomic orbital (AO) basis via the 4-index transformation,

$$v_{cd}^{ab} = C_p^a C_r^b C_c^q C_d^s < pq | \frac{1}{r} | rs > \tag{19}$$

The coefficients C in Eq. (19) are obtained from the iterative Hartree-Fock procedure. The transformed integrals have the following general symmetries,

$$v_{ar}^{bs} = v_{br}^{as}$$

$$v_{ab}^{rs} = v_{ba}^{sr}$$

The (T) correction is given as:

$$E^{[T]} = \bar{t}_{abc}^{ijk} t_{ijk}^{abc} D_{ijk}^{abc} \tag{20}$$

$$E^{(T)} = E^{[T]} + t_{ijk}^{abc} D_{ijk}^{abc} \bar{z}_{abc}^{ijk} \tag{21}$$

An arbitrary quantity $\bar{x}_{abc}^{ijk}$ is defined as

$$\bar{x}_{abc}^{ijk} = \frac{4}{3} x_{abc}^{ijk} - 2x_{acb}^{ijk} + \frac{2}{3} x_{bca}^{ijk} \tag{22}$$

and

$$z_{abc}^{ijk} = (t_a^i v_{bc}^{jk} + t_b^j v_{ac}^{ik} + t_c^k v_a^{ij} b) / D_{ijk}^{abc} \tag{23}$$

The T_3 amplitudes are,

$$D_{ijk}^{abc} t_{ijk}^{abc} = P(ia/jb/kc)[t_{ij}^{ae} v_{ek}^{bc} - t_{im}^{ab} v_{jk}^{mc}] \tag{24}$$

where the symmetrizer $P(ia/jb/kc)$ is

$$P(ia/jb/kc) u_{abc}^{ijk} = u_{abc}^{ijk} + u_{acb}^{ikj} + u_{bac}^{jik} + u_{bca}^{jki} + u_{cba}^{kji} + u_{cab}^{kij}$$

2. Computational details

The CCSD equations are non-linear and must be solved to self-consistency via an iterative procedure, usually with the help of an acceleration method [16]. The CCSD method is dominated by its most expensive term, $v_{ef}^{ab} c_{ij}^{ef}$, which scales as $v^4 o^2$, where *v*, *o* are the number of virtual and occupied molecular orbitals, respectively. Formally, the method is expensive in terms of memory and storage as well, with amplitude storage on the order of $v^2 o^2$ and integral storage on the order of $v^4, v^3 o, \ldots$ and so on. The amount of in-core memory depends on the specific algorithm used; most algorithms require $v^2 o^2$ storage per node. This amount of memory is not scalable. For example, a problem with 100 occupied and 1000 virtual orbitals would require 80GB of memory per node, which

is not commonly available.

The non-iterative (T) correction requires v^3o storage and scales as v^4o^3. A naive (T) algorithm is trivial to implement but an algorithm that has a small memory requirement and scalable I/O is more challenging.

There is a CCSD(T) method in nearly every quantum chemistry package. The ACES [17] and NWChem [18] implementations can handle very large computations, provided that a supercomputer is available [19]. The MOLPRO [20] algorithm has an o^2v^2 memory requirement, which limits its utility, but it is perhaps the fastest algorithm for smaller calculations. The GAMESS [21] implementation runs in parallel but is similarly limited by an o^2v^2 memory requirement. The Janowski, Ford, Pulay disk array CC implementation [22] can handle large computations of the order of a thousand basis functions on a commodity cluster by utilizing a filesystem for storage, but the performance of their algorithm is limited by disk I/O.

3. Design of a Scalable and Efficient Algorithm

In a previous paper, an MP2 energy algorithm was discussed [23], which has a small memory footprint, good performance, a flexible storage implementation, and is able to run on workstations and clusters equally well. In the same spirit, a coupled cluster algorithm can be designed, such that it is efficient, has a small memory footprint, is able to utilize a filesystem and memory for storage, and as a result can run on machines with very different capabilities.

For coupled cluster algorithms (and other many-body methods), it is the memory that is most likely to limit the application of the algorithm. Memory is a limited resource, unlike the time. Furthermore, the time to completion for calculations can be decreased by providing more computational hardware, whereas the amount of physical memory per node cannot be increased by adding another node.

Some very large arrays can (and need to) be distributed across the nodes (distributed memory) or stored on the filesystem. Disks are inexpensive and offer terabytes of storage, but filesystem I/O can be very slow if not done right. Nevertheless, a considerable amount of memory must be present to carry out local calculations.

What are the memory limitations of current hardware? A "*typical*" workstation or a cluster node in most research groups has between 1GB and 8GB of memory per core, with 2GB of RAM probably the most common. For an entire *node*, the amount of memory can be as much as 64GB or more, depending on the number of cores/node. That number will increase in the future, but possibly at a slower rate than the increase in computational power.

To draw a connection between memory and the dimensions present in CC calculations, several generic arrays of varying dimensions, corresponding to 100 occupied orbitals and 1000 and 2000 basis functions, are listed in Table 1. The dimensions of these arrays may correspond, for example, to an entire integral array or to the first three indices. The algorithm design is then guided by what arrays are small enough to be stored per node or per core. It should be kept in mind that the sizes listed are not for the entire calculation, but for one of the several arrays needed. Some of the arrays can be shared, but some must be allocated per thread/core.

Storing an o^2n^2 array per node (let alone per core) is too expensive: A node with 80GB of RAM is rare and one with 320GB is even more rare. The same is true for the quartic arrays other than o^4 and arrays involving an n^2 factor. Storing, for example, several 3GB arrays would preclude most systems from being able to handle more than a thousand basis functions. The choice is then to restrict memory requirements to o^2n (or smaller) arrays, whose size only increases linearly with basis set. Trying to limit memory further than o^2n, to say on, will come at a very high cost of increased I/O.

Table 1. Array Sizes for o=100

Array	Size (GB), n_{basis}=1000	Size (GB), n_{basis}=2000
o^4	0.8	0.8
o^2n	0.08	0.16
on^2	0.8	3.2
o^3n	8.0	16.0
n^3	8.0	64.0
o^2n^2	80.0	320.0
on^3	800.0	6400.0
n^4	8000.0	128000.0

Some arrays, notably n^4, are too great to store even in secondary storage. The terms involving such an array must be evaluated directly, i.e. on the fly, at the modest cost of recomputing atomic integrals, cf. Olson *et al* [24]. However, to push the ability of the algorithm beyond a thousand basis functions, on^3 storage also must be eliminated in the CCSD algorithm. To ensure that I/O overhead is low even on filesystems, transfers to and from secondary storage must be contiguous and in large chunks. There are three basic remote operations: `put`, `get`, `accumulate`. The last of these cannot be implemented efficiently via the filesystem I/O and the algorithm must not rely on it.

Finally, to achieve computational efficiency, all of the expensive tensor contractions that must be carried out using `dgemm` and tensor permutations must not exceed two adjacent indices to ensure data locality**,** e.g., A(j,i,k) = A(i,j,k) is OK, but A(k,j,i) = A(i,j,k) is not, because the latter has poor memory performance. The work distribution between the nodes must be over the virtual index rather than the (usually) much smaller occupied index, to ensure that the algorithm can scale to hundreds of nodes. The work within the node can be parallelized using threads. This multi-level parallelization guarantees that the algorithm will scale to thousands of cores.

In the following discussion, the primary focus is on memory, then on secondary storage and I/O, and only then on the computational aspect. The consequent performance is illustrated below with benchmarks.

4. Implementation

This section is broken into three sub-sections that address the direct CCSD terms, the non-direct CCSD terms, and the triples correction, respectively. The CCSD component of the CCSD(T) algorithm is by far the most complex due to the number of terms.

Before proceeding to the respective sections, consider I/O optimization via loop blocking. In Algorithm 1, B is a blocking factor. If $B = 1$, then it is just a regular loop: the innermost (most expensive) load operation is executed N^3 times, the total I/O overhead is M^2N^3, and the local buffer size is M^2. If B is greater than 1, the innermost load operation is called $(N/B)^3$ times, the I/O overhead is $M^2B(N/B)^3 = M^2N^3/B^2$, and the local buffer size is M^2B. So, at the cost of increasing the local buffer size, the I/O overhead can be reduced by a factor of B^2. In general, loop blocking decreases I/O by $B^{(L-1)}$ where L is the number of nested loops.

The loop blocking will be used where I/O might pose a problem. Since blocking also requires an increase in memory overhead, the blocking factor can be determined by setting a runtime memory limit.

```
for i = 0:N,B { // iterate to N in steps of B
  for j = 0:N,B {
    for k = 0:N,B {
      // the innermost load operation
      buffer(M,M,B) = load A(M,M,k:k+B)
      ...
    }
  }
}
```

Algorithm 1. Loop Blocking

4.1. Direct Terms

As mentioned already, v_{cd}^{ab} has to be evaluated directly due to storage constraints. The same approach can be extended to evaluate terms v_{bc}^{ia} directly as well at little additional cost.

To make the notation simpler, the conventional VT notation is used, where the general

single and double amplitudes contractions are referred to as VT_1, VT_2, VT_1^2, the latter implying contraction with two single amplitudes.

The integral v_{cd}^{ab} is contracted with $T_{1\ ij}^{2\ cd} = (T_1 T_1)_{ij}^{cd} = t_i^c t_j^d$ and $T_{2\ ij}^{\ cd} = t_{ij}^{cd}$ amplitudes,

$$VT_{2\ ij}^{\ ab} = t_{ij}^{cd} C_d^s C_c^q V_{qs}^{pr} C_r^b C_p^a \tag{25}$$

$$VT_{1\ ij}^{2\ ab} = t_i^c t_j^d C_d^s C_c^q V_{qs}^{pr} C_r^b C_p^a \tag{26}$$

Half-transforming the amplitudes to the AO basis and factoring out half-contracted terms yields expressions in terms of half-transformed intermediates U, with subscripts referring to the T contraction (Recall that p,q,r,s are AO indices.).

$$U_{2\ ij}^{\ pr} = (t_{ij}^{cd} C_d^s C_c^q) V_{qs}^{pr} \tag{27}$$

$$U_{1\ ij}^{2\ pr} = (t_i^c C_c^q)(t_j^d C_d^s) V_{qs}^{pr} \tag{28}$$

$$VT_{2\ ij}^{\ ab} = U_{2\ ij}^{\ pr} C_r^b C_p^a \tag{29}$$

$$VT_{1\ ij}^{2\ ab} = U_{1\ ij}^{2\ pr} C_r^b C_p^a \tag{30}$$

All similar VT terms can be obtained from U at virtually no cost by having the last two AO indices transformed to occupied and virtual indices. For example, the v_{bc}^{ia} terms in Equation (7) are just

$$v_{ef}^{ma}(2t_{mi}^{ef} - t_{im}^{ef}) = 2U_{mi}^{qs} C_q^m C_s^a - U_{im}^{qs} C_q^m C_s^a \tag{31}$$

The v_{bc}^{ia} also enter the VT_1 diagrams,

$$VT_{1\ ab}^{\ ij} = t_i^c C_j^s C_c^q V_{qs}^{pr} C_r^b C_p^a \tag{32}$$

$$VT_{1\ jb}^{\ ia} = t_i^c C_d^s C_j^q V_{qs}^{pr} C_r^b C_p^a \tag{33}$$

and two more intermediates are needed,

$$U_{1\ qs}^{\ ij} = (t_a^i C_p^a) C_r^j V_{qs}^{pr} \tag{34}$$

$$U_{1\ js}^{\ ir} = (t_a^i C_p^a) C_j^q V_{qs}^{pr} \tag{35}$$

which can then be transformed into appropriate VT_1 diagrams.

Now, if all four U intermediates are available, neither v_{cd}^{ab} nor v_{bc}^{ia} need to be stored for the CCSD iterations; they can be replaced with much smaller $4o^2n^2$ storage.

Half-transformed T_2 amplitudes, Eq. (27), also provide a way to devise a direct contraction algorithm with very little memory requirement. Since the contraction is in the AO basis, atomic indices can be contracted without having to construct V_{qs}^{ab} which would require all atomic basis p,r indices and thus N^2M^2 memory, where M is the size of the largest shell. Algorithm 2 only needs NM^3 memory.

```
for S in Shells {
  for Q ≤ S {
    for R in Shells {
      for P in Shells {
        // skip insignificant ints
        if (!screen(P,Q,R,S)) continue;
        // evaluate 2-e integrals(PQ|RS)
        G(P,R,Q,S) = eri(P,Q,R,S);
      }
      for r in R {
        U1(i,j,q,s) = ...
        U12(i,j,q,s) = ...
        load t(o,o,n,r)
        U2(i,j,q,s) += t(i,j,p,r)*G(p,r,q,s)
      }
    }
    store U1(i,j,Q,S), U1(j,i,S,Q)
    store U12(i,j,Q,S), U12(j,i,S,Q)
    store U2(i,j,Q,S), U2(j,i,S,Q)
  }
}
```

Algorithm 2. Direct CCSD intermediates

The important points of Algorithm 2 are:

- The integral symmetry is exploited to halve the number of integral calculations *and* transformations.
- The loop over Q,S can be distributed over nodes.
- The loop over R can be parallelized over threads. In this case, the U storage can be shared, provided the updates to shared memory are synchronized.

- The innermost $t2$ loads can be reduced by blocking the Q, S loops (cf. the discussion on loop blocking).
- Per thread storage is NM^3, which is 16MB for a basis set of size 2000 with f shells ($M = 10$). The local U storage is likewise small, only 8MB for $o = 100$. This tiny memory footprint allows for a very large Q, S blocking factor and consequently the I/O can be dramatically reduced.

Note that both UT_1 terms cannot be evaluated simultaneously using the above algorithm, as they correspond to two different integrals, $< pq | rs >$ and $< pr | qs >$. However, one of them can easily be evaluated by applying the algorithm a second time to compute a single UT_1 term at a very modest on^4 computational cost.

4.2. CCSD

Because the singles amplitudes storage is negligible, on, the singles part of the CCSD code is easy to implement and parallelize. By making a virtual index the outermost index, the local memory is guaranteed not to exceed o^2n since all of the diagrams with three and four virtual indices have already been evaluated above.

The doubles amplitudes calculation requires the most effort to implement, primarily due to the number of contractions and the terms that require significant I/O. Recall that all v_{cd}^{ab} and v_{bc}^{ia} terms have been evaluated, as have many similar VT terms.

The first step towards deriving a scalable algorithm for Dt_{ab}^{ij} (See Eq. (8)) is to fix the outermost loop at the outermost virtual index *b*, since the *b* index can be evaluated across nodes independently. For each *b* iteration an o^2n Dt_{ab}^{ij} block is evaluated and stored.

The quantities with a b index are loaded once, guaranteed not to exceed size o^2n. The tensors without a b index imply that the tensor is needed in its entirety for each b iteration. To ensure that no v or t memory exceeds o^2n, those tensors without a b index must be loaded into memory o^2n tiles at a time for each b index inside a loop over a dummy virtual orbital index, u. This increases the I/O cost to o^2n^2 per *b* index, or o^2n^3 overall, which is still below the o^3n^3 computational cost.

There are three tensors that must be contracted fully for a given b index: $v_{ia}^{jb}, v_{ij}^{ab}, t_{ij}^{ab}$. The loop corresponding to v_{ia}^{jb} can be eliminated right away, it is only needed in its entirety to evaluate $I_{ie}^{ma} t_{mj}^{eb}$ in Eq. (8). Since, this term appears inside the symmetrizer P,

$$P(v_{je}^{mb} t_{mi}^{ea}) = P(v_{ie}^{ma} t_{mj}^{eb})$$

I_{ie}^{ma} can be replaced by an equivalent I_{je}^{mb}. This leads to Algorithm 3.

```
for b in v {  //  loop  over  virtual  b
index
  Dt(i,j,a) = 0

  load t(o,o,v,b)
  load V(o,o,v,b)
  load V(o,v,o,b)
  load V(o,o,o,b)

  Dt += Vt

  // terms with t
  for u in v {
    load t'(o,o,v,u)
    // evaluate terms with t'
    Dt += Vt'
  }

  // terms with v
  for u in v {
    load v'(o,o,v,u)
    // evaluate terms with v'
    Dt += V't
  }

  store Dt(o,o,v,b)
}
```

Algorithm 3. CCSD

The important points about Algorithm 3:

- The loop over the b index is easy to make parallel.
- The local memory is on the order $4o^2n$ plus o^2n per innermost v'/t'

temporary storage, corresponding to loading all of the v_{ij}^{ab}, t_{ij}^{ab} quantities, one virtual index at a time**.**

- The b loop can be easily blocked to reduce the I/O by a blocking factor B at the expense of increasing the memory by a factor of B. .
- Since the memory footprint is low, B can be fairly large. For example, for O=100, V=2000, B=4 and B=8, the required memory is 2.6 GB and 5.2 GB per *node*, respectively.
- The operations outside the u loop can be parallelized inside the node by using a threaded math library.
- The operations inside the u loop can be explicitly parallelized inside the node via threads, with the added benefit of overlapping I/O and computations.

4.3. (T)

The (T) correction, Eq. (14), only involves t_{ab}^{ij}, v_{ka}^{ij}, v_{ab}^{ij}, and v_{bc}^{ia}. The unused CCSD arrays previously allocated can be freed to make space for v_{bc}^{ia}. Since v_{bc}^{ia} was never constructed, another integral transformation needs to be carried out at a small on^4 cost.

The Piecuch (T) correction [15] equations were given in a way that requires keeping an occupied index fixed and permuting the virtual index. In other words the local memory required for t_{abc}^{ijk} would have been v^3. Since the triples amplitudes are symmetric with respect to the exchange of index "columns",

$$t_{abc}^{ijk} = t_{bac}^{jik} = t_{acb}^{ikj} = \ldots$$

all terms with t_{bac}^{jik} can be written with the virtual index fixed, e.g., $t_{bac}^{ijk} = t_{abc}^{jik}$, $t_{cab}^{ijk} = t_{abc}^{jki}$, etc.

Now the T_3 amplitudes can be implemented as a series of 12 `dgemms` and 6 index permutations, as illustrated in Algorithm 4. The important points about Algorithm 4 are:

- The symmetry in a, b, c indices is utilized.
- The loop over a, b, c indices is easily parallelizable.
- Only the loads with an a index are innermost
- The loops can be easily blocked to reduce the I/O by a factor of B^2 where

B is the blocking factor.

- The local storage required is $3o^2vB + 3o^3B + 6ovB^2 + o^3B^3$
- If $B > 1$, the actual `dgemms` are carried out inside another B^3 loop, which can be parallelized *within* a node by using threads.
- Since the memory footprint is low, the blocking factor can be large. For example, for O=100, V=1000, B=4 and B=8, the required memory is 1.6G and 6.4G per *node* respectively.

```
for c in V {
  for b in c {
    for a in b {

      load t(o,o,a,b)
      load t(o,o,a,c)
      load t(o,o,b,c)

      load v(o,o,o,a)
      load v(o,o,o,b)
      load v(o,o,o,c)

      load v(o,o,v,a)
      load v(o,o,v,b)
      load v(o,o,v,c)

      load v(o,v,b,c)
      load v(o,v,c,b)
      load v(o,v,a,c)
      load v(o,v,c,a)
      load v(o,v,a,b)
      load v(o,v,b,a)

      // t(i,j,e,a)*V(e,k,b,c) corresponds to
      // dgemm(t(ij,e), V(e,k)), etc
      t(i,j,k) = t(i,j,e,a) V(e,k,b,c) - t(i,m,a,b) V(j,k,m,c)
      t(i,k,j) = t(i,k,e,a) V(e,j,c,b) - t(i,m,a,c) V(k,j,m,b)
      t(k,i,j) = t(k,i,e,c) V(e,j,a,b) - t(k,m,c,a) V(i,j,m,b)
      t(k,j,i) = t(k,j,e,c) V(e,i,b,a) - t(k,m,c,b) V(j,i,m,a)
      t(j,k,i) = t(j,k,e,b) V(e,i,a,c) - t(j,m,b,c) V(k,i,m,a)
      t(j,i,k) = t(j,i,e,b) V(e,k,c,a) - t(j,m,b,a) V(i,k,m,c)
      ...
    }
  }
}
```

Algorithm 4. (T)

4.4. The overall picture.

The algorithm is implemented entirely in C++, as a part of stand-alone library (LIBCCHEM) which includes previously reported ERI (electron repulsion integrals), Fock, and MP2 methods [23,25,26]. The library requires only minimal input from the host program and can be connected to a variety of packages.

The storage is implemented using Global Arrays (GA) [27] for distributed memory and HDF5 [28] for file storage, since the GAMESS distributed memory interface (DDI) [29] does not currently support arrays of more than 2 dimensions. The arrays are first allocated in faster GA memory until the limit is reached, and then on the filesystem. The arrays responsible for the most I/O need to be allocated first to ensure that they reside in distributed memory.

The overall algorithm may be outlined as follows:

- The CCSD arrays are allocated, with t and v_{ij}^{ab} first to ensure that these arrays are in fast storage. Overall, storage is needed for t, v_{ij}^{ab}, v_{ij}^{ka}, v_{ia}^{jb}, v_{ij}^{kl}, Dt and four U intermediates
- The allocated arrays are evaluated using the regular 4-index transformation.
- The initial $T2$ amplitudes are taken to be the MP2 amplitudes, $v_{ij}^{ab} / D_{ij}^{ab}$, and the $T1$ amplitudes are set to zero.
- The intermediate U storage is allocated.
- The CCSD equations are repeated until an acceptable threshold is reached, either the energy difference or the amplitude difference.
- The CCSD step is optionally accelerated using DIIS [16].
- Once converged, all but the first three arrays are freed and v_{ia}^{bc} array is allocated and evaluated.
- The non-iterative (T) method is performed.

5. *Performance*

To assess the performance and applicability of the algorithm, three scenarios are considered here: single node performance, performance on a cluster of modest size, and high-end cluster performance. The inputs are selected to reflect a range of basis functions and occupied orbitals.

The modest cluster, Exalted, is composed of nodes connected by InfiniBand. Each node has one Intel X5550 2.66GHz 6-core processor, 24GB of RAM, two local disk drives, and an NVIDIA Fermi C2050 GPU card.

First, consider the ability of the algorithm to run on a single node and to use a filesystem in case not enough memory is available to store all data, Table 2. As can be seen, even on a single node, fairly large CCSD(T) jobs can still run in a reasonable timeframe (i.e., less than a week). Despite falling back to disk in all cases, across the board the I/O time as a fraction of total time is very small, below 5%.

Table 2. Exalted Single Node Performance.

Input	#AO/Occ[1]	CCSD[2]	(T)	(T) Mem/Disk[3]	(T) I/O
$C_4N_3H_5$/aug-ccPVTZ	565/21	42m	8h	2.1/19.5 GB	13m
$C_8H_{10}N_4O_2$/aug-ccPVDZ	440/37	50m	17h	5.5/17.0 GB	13m
$SiH_4B_2H_6$/aug-ccPVQZ	875/16	141m	18h	3.4/53.4 GB	49m
$C_8H_{10}N_4O_2$/ccPVTZ	640/37	180m	64h	12.2/49.0 GB	42m

[*] m refers to minutes, h refers to hours
[1] Number of atomic/occupied orbitals
[2] single CCSD iteration time
[3] Memory/Disk used to evaluate (T)

The cluster performance is assessed on the basis of the time larger jobs take to run, Table 2, and the scalability of a medium-size job, Table 3. First, all of the inputs used for single node benchmarking can run in under a day on the cluster. Secondly, a large CCSD Tamoxifen calculation, C26H29NO, can run on this relatively small (Exalted) cluster, three hours per iteration.

As expected, the (T) algorithm scales well, as shown in Table 4, since it is very easy to parallelize to a large number of nodes. However, the scalability of the CCSD algorithm is

not perfect. This is especially noticable when running on a large cluster, such as the Cray XE6 system, which has thousands of cores the two 16-core AMD Bulldozer nodes, with 64GB of RAM, connected by a fast network. The performance gain from increasing the number of nodes, Table 5, is below linear scaling, but the longer Tamoxifen calculation scales reasonably well to 1024 cores, reducing the runtime by a factor of 3.3 relative to the 256 core run.

Each XE6 node has two chips, 16 cores each. The benchmarks in Table 5 were obtained running 32 threads over the entire node created from a single MPI process. The better option, especially in the case of (T) is to run one MPI process per *chip* rather than per *node*, as illustrated in Table 6. If each MPI process runs (and creates threads) within a single chip only, the threads do not need to communicate over the slower bridge connecting two chips. Generally, there is a large penalty for sharing data across the *chips*, which must be avoided by having a flexible approach to launch jobs.

Table 3. Exalted Cluster Performance. All times are in minutes.

Input	#AO/Occ[1]	# cores	CCSD[2]	(T)
$C_4N_3H_5$/aug-ccPVTZ	565/21	24	12	61
$SiH_4B_2H_6$/aug-ccPVQZ	875/16	48	20	133
$C_8H_{10}N_4O_2$/ccPVTZ	640/37	48	26	482
$C_{26}H_{29}NO$/aug-ccPVQZ	961/71	96	211	N/A[3]

[1] Number of atomic/occupied orbitals
[2] single CCSD iteration time
[3] Job requires 0.5TB of storage: Exalted does not have sufficient memory or parallel FS.

Table 4. Exalted Cluster Scaling, $C_8H_{10}N_4O_2$/cc-PVTZ. All times are in minutes.

Cores/Nodes	CCSD[1]	(T)
24/4	28	971
48/8	15	482
96/16	11	240

[1] single iteration time

Table 5. Cray XE6 CCSD Performance. All times are in minutes per single iteration.

# cores	256 cores	512 cores	1024 cores
$SiH_4B_2H_6$ (T)/aug-ccPVQZ	130	76	42
$C_8H_{10}N_4O_2$ CCSD/ccPVTZ	15	9	6
$C_{26}H_{29}NO$ CCSD/aug-ccPVQZ	253	134	76

Table 6. Cray XE6 Intra-Node Configuration, $SiH_4B_2H_6$ (T)/aug-ccPVQZ. All times are in minutes.

# cores	32x1 Threads/MPI	16x2 Threads/MPI
256	130	101
512	76	49
1024	42	27

4.1. GPU CCSD Performance

As expected, the direct terms account for the most time in CCSD iterations. In the present implementation most of that work is concentrated in a continuous application of just one `dgemm` operation. Adding a graphical processor (GPU) `dgemm` to handle matrix multiplication, while keeping the integral evaluation on the host, is fairly easy. In a multithreaded environment, several threads must be assigned to a GPU device to avoid work imbalance.

Augmented with GPU BLAS, via CUBLAS [30], the CCSD calculations on a single exalted node get a noticeable speed up, shown in Table 7, if the direct term (See Section 4.1) dominates the entire iteration (this is the case if the number of occupied orbitals is very small relative to the size of the basis set). If the number of occupied orbitals is relatively high, the direct term accounts for a smaller fraction of the total iteration time, and consequently the GPU benefit is less noticeable overall. At the time, the (T) GPU implementation is not complete.

Table 7. Exalted Single Node+GPU CCSD performance. All times are in minutes per iteration.

Input	$C_8H_{10}N_4O_2$/ccPVTZ	$SiH_4B_2H_6$/aug-ccPVQZ	$C_4N_3H_5$/aug-ccPVTZ
Direct	124	131	36
Direct+GPU[1]	53	65	26
CCSD	163	142	42
CCSD+GPU[1]	115	75	33
CCSD Speed-up[2]	1.4x	1.9X	1.3X

[1] GPU enabled

[2] Overall CCSD speed-up relative to CPU code

5. Conclusions

The algorithm presented in this paper is able to handle fairly large jobs on a single node, a small cluster, and high-end Cray system. The algorithm has a small adjustable memory footprint and is able to optionally use the filesystem if the data exceeds distributed memory storage. The algorithm can also optionally use GPUs to speed up certain CCSD computations. When running on the multi-core node with multiple processor packages (chips), the algorithm benefits from limiting thread communication to within a chip.

The algorithm is implemented entirely in C++, as a part of stand-alone library which includes previously reported ERI, Fock, and MP2 methods. [18,19].

References

[1] JA Pople, R. Seeger, and R. Krishnan. Variational configuration interaction methods and comparison with perturbation theory. *International Journal of Quantum Chemistry*, 12(S11):149–163, 1977.

[2] J.A. Pople, M. Head-Gordon, and K. Raghavachari. Quadratic configuration interaction. a general technique for determining electron correlation energies. *The Journal of chemical physics*, 87(10):5968–5975, 1987.

[3] J.A. Pople, P.M.W. Gill, and B.G. Johnson. Kohn-Sham density-functional theory within a finite basis set. *Chemical physics letters*, 199(6):557–560, 1992.

[4] F. Coester and H. Kümmel. Short-range correlations in nuclear wave functions. *Nuclear Physics*, 17:477–485, 1960.

[5] J. Cizek. On the correlation problem in atomic and molecular systems. Calculation of wavefunction components in ursell-type expansion using quantum-field theoretical methods. *The Journal of Chemical Physics*, 45(11):4256-4266, 1966.

[6] J. Paldus, J. Cizek, and I. Shavitt. Correlation problems in atomic and molecular systems. IV. Extended coupled-pair many-electron theory and its application to the BH3 molecule. *Physical Review A*, 5(1):50, 1972.

[7] D. Mukherjee, R.K. Moitra, and A. Mukhopadhyay. Applications of a nonperturbative many-body formalism to general open-shell atomic and molecular problems: calculation of the ground and the lowest pi-pi* singlet and triplet energies and the first ionization potential of trans-butadiene. *Molecular Physics*, 33(4):955-969, 1977.

[8] I. Lindgren. A coupled-cluster approach to the many-body perturbation theory for open-shell systems. *International Journal of Quantum Chemistry*, 14(S12):33-58, 1978.

[9] G.E. Scuseria, C.L. Janssen, and H.F. Schaefer. An efficient reformulation of the closed-shell coupled cluster single and double excitation (CCSD) equations. *The Journal of Chemical Physics*, 89(12):7382-7387, 1988.

[10] G.D. Purvis III and R.J. Bartlett. A full coupled-cluster singles and doubles model: The inclusion of disconnected triples. *The Journal of Chemical Physics*, 76:1910, 1982.

[11] K. Raghavachari, G.W. Trucks, J.A. Pople, and M. Head-Gordon. A fifth-order perturbation comparison of electron correlation theories. *Chemical Physics Letters*, 157(6):479–483, 1989.

[12] P. Piecuch, S.A. Kucharski, and R.J. Bartlett. Coupled-cluster methods with internal and semi-internal triply and quadruply excited clusters: CCSDT and CCSDTQ approaches. *The Journal of chemical physics*, 110:6103, 1999.

[13] A. Szabo and N.S. Ostlund. Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory. Dover Books on Chemistry Series. Dover Publications, 1996.

[13] D. Yarkony. *Modern Electronic Structure Theory*. Number pt. 2 in Advanced Series in Physical Chemistry. World Scientific, 1995.

[14] I. Shavitt and R.J. Bartlett. *Many-Body Methods in Chemistry and Physics: MBPT and Coupled-Cluster Theory*. Cambridge Molecular Science. Cambridge University Press, 2009.

[15] P. Piecuch, S.A. Kucharski, K. Kowalski, and M. Musiał. Efficient computer implementation of the renormalized coupled-cluster methods: The R-CCSD [T], R-CCSD (T), CR-CCSD [T], and CR-CCSD (T) approaches. *Computer Physics Communications*, 149(2):71–96, 2002.

[16] G.E. Scuseria, T.J. Lee, and H.F. Schaefer. Accelerating the convergence of the coupled-cluster approach: The use of the DIIS method. *Chemical physics letters*, 130(3):236–239, 1986.

[17] V. Lotrich, N. Flocke, M. Ponton, AD Yau, A. Perera, E. Deumens, and RJ Bartlett. Parallel implementation of electronic structure energy, gradient, and hessian calculations. *The Journal of chemical physics*, 128:194104, 2008.

[18] DE Bernholdt, E. Apra, HA Früchtl, MF Guest, RJ Harrison, RA Kendall, RA Kutteh, X. Long, JB Nicholas, JA Nichols, et al. Parallel computational chemistry made easier: The development of NWChem. *International Journal of Quantum*

Chemistry, 56(S29):475–483, 1995.

[19] PCC benchmarks. http://www.qtp.ufl.edu/PCCworkshop/PCCbenchmarks.html.

[20] H.J. Werner, PJ Knowles, R. Lindh, FR Manby, M. Schütz, P. Celani, T. Korona, G. Rauhut, RD Amos, A. Bernhardsson, et al. Molpro, version 2006.1, a package of Ab Initio programs. 2006.

[21] M. S. Gordon and M. W. Schmidt. *Advances in electronic structure theory: GAMESS a decade later*, pages 1167–1189. Elsevier, Amsterdam, 2005.

[22] T. Janowski, A.R. Ford, and P. Pulay. Parallel calculation of coupled cluster singles and doubles wave functions using array files. Journal of Chemical Theory and Computation, 3(4):1368–1377, 2007.

[23] A. Asadchev and M.S. Gordon. A new algorithm for second order perturbation theory. *Journal of Chemical Theory and Computation*, submitted

[24] R.M. Olson, J.L. Bentz, R.A. Kendall, M.W. Schmidt, and M.S. Gordon. A novel approach to parallel coupled cluster calculations: Combining distributed and shared memory techniques for modern cluster based systems. *Journal of Chemical Theory and Computation*, 3(4):1312–1328, 2007.

[25] A. Asadchev and M.S. Gordon. New multithreaded hybrid CPU/GPU approach to Hartree-Fock. *Journal of Chemical Theory and Computation*, in press.

[26] A. Asadchev, V. Allada, J. Felder, B.M. Bode, M.S. Gordon, and T.L. Windus. Uncontracted Rys Quadrature implementation of up to g functions on graphical processing units. Journal of Chemical Theory and Computation, 6(3):696–704, 2010.

[27] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–231, 2006.

[28] The HDF Group. Hierarchical data format, Version 5. http://www.hdfgroup.org/HDF5.

[29] R.M. Olson, M.W. Schmidt, M.S. Gordon, and A.P. Rendell. Enabling the efficient use of SMP clusters: the GAMESS/DDI model. In Proceedings of the 2003 ACM/IEEE conference on Supercomputing, page 41. ACM, 2003.

[30] R. Nath, S. Tomov, and J. Dongarra. Accelerating GPU kernels for dense linear algebra. *High Performance Computing for Computational Science–VECPAR 2010*, pages 83–92, 2011.

Chapter 5. Conclusions

As the computing technology changes and matures, scientific computing must follow.
Hardware and software that was cutting edge in the 70's and 80's still dictates how many of the computational chemistry packages are implemented today. However, computing technology evolved very quickly since the introduction of Fortran 77. Object oriented programming (OOP), generic programming, standard libraries, and system standards have become the essential pieces of most modern commercial and open-source software, small and large alike. To keep up with the improvements in computer science, computational chemistry algorithms must be either modernized or rewritten. Often, due to software architecture decisions made decades ago, rewriting is the only viable plan for the future. Not all of the software needs to be modernized at once: the key pieces such as integral and Hartree-Fock methods can be rewritten alone and integrated into the existing software, one at a time.

Software modernization also presents an opportunity to improve the existing algorithms, separate them into modular libraries to encourage reuse among the scientists, and to plan ahead, given the trends in computing over the last few decades.

The first algorithm presented was for the Hartree-Fock method, the reference method in the most electron correlation theories. The Hartree-Fock method requires

evaluation of the two-electron integrals, which constitutes the most consuming part. Unlike other pieces in computational chemistry, two-electron integral methods are specific to the domain and do not receive much attention from outside the field. In the present work the integrals were implemented using the Rys Quadrature approach, one of several integral methods. While algorithmically more complex than other methods, the Rys Quadrature method is a general numerically stable method with low memory footprint, which makes it suitable for implementation on graphical processing units (GPU).

Once the integral engine was implemented, the multithreaded Hartree-Fock method naturally followed. The integral and Hartree-Fock GPU implementation was able to reuse many key pieces of the CPU algorithm, designed to be fast, extensible, and flexible through the use of a code generator and C++ templates.

One of the most common electron correlation methods is second order many-body perturbation theory (MBPT2), also known as Moller-Plesset second order perturbation theory (MP2). Unlike higher-order treatments, MP2 is a relatively inexpensive black-box method which makes it very popular. Hence, the Hartree-Fock implementation was followed by an implementation of the MP2 method. Like the Hartree-Fock method, the MP2 implementation relies heavily on fast integrals. But unlike Hartree-Fock, most of computational work is handled by the de facto standard basic linear algebra subroutines, BLAS. The MP2 algorithm implemented is a semi-direct method, meaning that the partially transformed integrals need to be stored in secondary storage, such as disk or distributed memory. Unlike the other MP2 algorithms, which are based on either disk or distributed memory, the implemented algorithm uses Object Oriented Programming (OOP) features of C++ to provide transparent integral storage on either disk or in distributed memory.

The natural follow-up to MP2 is coupled cluster (CC) theory. The coupled cluster method, truncated at singles and doubles excitations, CCSD, with a perturbative triples correction (T) leads to the CCSD(T) method, often called the gold standard of computational chemistry due its accuracy. CCSD(T) is very expensive method, both in terms of computer time and memory. However, with the lessons learned designing the

MP2 algorithm, a fast
CCSD(T) algorithm was developed such that it could run on both a single workstation and supercomputers. The key to the implementation was optimizing the algorithm in terms of memory first, I/O overhead second, and concentrating on the computational efficiency last.
By using several properties of atomic to molecular basis transformations, several expensive computation and storage requirements were eliminated from the CCSD algorithm. And by using the well-known loop optimization technique called blocking, the (T) algorithm was implemented with very little memory requirement and very little I/O overhead.

The three algorithms summarized above were prompted by the need to accommodate the wide array of computational hardware. In the process, the algorithms were improved, often drastically. Implemented in C++, the algorithms and the supporting framework were built as a stand-alone library, with Fortran bindings. Connected to GAMESS, the library was successively integrated with the existing legacy code. While not explicitly discussed, the supporting framework, such as basis set and wavefunction objects, is absolutely necessary to develop robust flexible modern code.