JCTC Journal of Chemical Theory and Computation

A Novel Approach to Parallel Coupled Cluster Calculations: Combining Distributed and Shared Memory Techniques for Modern Cluster Based Systems

Ryan M. Olson,[†] Jonathan L. Bentz,[‡] Ricky A. Kendall,[‡,§] Michael W. Schmidt,[†] and Mark S. Gordon*,[†]

Department of Chemistry, Iowa State University, Ames, Iowa, Department of Computer Science, Iowa State University, Ames, Iowa 50011, and Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831

Received January 17, 2007

Abstract: A parallel coupled cluster algorithm that combines distributed and shared memory techniques for the CCSD(T) method (singles + doubles with perturbative triples) is described. The implementation of the massively parallel CCSD(T) algorithm uses a hybrid molecular and "direct" atomic integral driven approach. Shared memory is used to minimize redundant replicated storage per compute process. The algorithm is targeted at modern cluster based architectures that are comprised of multiprocessor nodes connected by a dedicated communication network. Parallelism is achieved on two levels: parallelism within a compute node via shared memory parallel techniques and parallelism between nodes using distributed memory techniques. The new parallel implementation is designed to allow for the routine evaluation of mid- (500−750 basis function) to large-scale (750−1000 basis function) CCSD(T) energies. Sample calculations are performed on five low-lying isomers of water hexamer using the aug-cc-pVTZ basis set.

I. Introduction

Coupled-cluster (CC) methods[1−3] are now widely accepted as the premier single-reference electronic structure methods for small chemical systems at or near equilibrium geometries. One of the most popular CC methods is CCSD(T), which is based on an iterative solution of the single and double (SD) cluster amplitude equations[4] with a noniterative perturbative correction for the triples (T).[5] The CCSD(T) approach has been shown[6] to be a good compromise between the chemical accuracy of the higher-order CCSDT (full triples) method[7] and the computational efficiency of low order many-body perturbation theory (MBPT). Equation of motion (EOM) CC methods[8−12] have been developed for excited-state calculations. Spin flip[13,14] and method of moments CC methods,[15] including the popular renormalized (R),[15] completely renormalized (CR),[15] and CR-CCSD(T)$_L$ (CCL) methods,[16] have extended formally single-reference CC methods into the regime of bond making and bond breaking, an area where traditional CC methods break down.

The biggest drawback of CC methods is the large computational demands required to perform such calculations. However, due to the popularity of methods like CCSD(T), considerable research has been carried out to generate highly efficient algorithms[4,17−21] and their implementations. A variety of CC methods can be found in all of the major electronic structure programs available today, including GAMESS,[22] MOLPRO,[23] ACES II,[24] Q-CHEM,[25] PSI3,[26] NWCHEM,[27,28] DALTON,[29] and GAUSSIAN03.[30] Most CC programs are highly optimized to run sequentially. This usually means the calculation is performed on a single processor. The speed of the processor and the size of the associated memory and disk are limiting factors for sequential algorithms. CCSD(T) calculations, especially those run in C_1 symmetry, reach the limit of most single processor workstations at around 400−500 basis functions (BF); even then, calculations of these sizes may require weeks of time on a dedicated workstation.[31]

* Corresponding author e-mail: mark@si.fi.ameslab.gov.
[†] Department of Chemistry, Iowa State University.
[‡] Department of Computer Science, Iowa State University.
[§] Oak Ridge National Laboratory.

One means of evaluating computationally demanding problems such as large basis set ($>$500 BF) CCSD(T) calculations is to make use of parallel computing. Parallel computing involves simultaneously evaluating multiple portions of a larger computational problem on multiple processors, in order to achieve an overall reduction in the real-time evaluation of the problem. Equally important, parallel computing can extend computationally demanding methods like CCSD(T) to larger problems because of increased computational resources and also storage (memory/disk) resources. There is a wide range of parallel computing environments and methodologies, two examples of which are addressed herein. These are as follows: (1) parallelism that is achieved by using multiple computers or *nodes* which are connected by a dedicated communication network and (2) parallelism that is achieved by multiple processors within a single *node* that share "local" system resources including memory and I/O channels.

The tools and methodologies for these two traditional types of parallel computing environments are very different. Multinode parallelism focuses on combining replicated and/or distributed memory techniques using parallel communication libraries such as TCGMSG,[32] SHMEM,[33] MPI,[34] Global Arrays (GA),[35] and the Distributed Data Interface (DDI).[36,37] One advantage of multinode models is that the aggregate system resources increase as the number of nodes increases, thereby facilitating more resource demanding calculations. However, since the nodes are distinct and internode communication must travel over a high-speed network, there are three factors that will strongly affect the performance for these types of calculations: (1) the performance (bandwidth and latency) of the network, (2) the total amount of internode communication required, and (3) the degree to which the necessary communication can be overlapped with computation. Single node multiprocessor parallel schemes have traditionally focused on a relatively small number of compute processes (or threads), usually between 2 and 16, using shared resources as a means to reduce (1) message passing communication and (2) replicated storage overhead, i.e., using the shared resources of the system to store certain data arrays only one time, rather than stored multiple identical copies for each process (or thread). A major focus of these techniques involves sharing portions of the system memory among all parallel processes (or threads) and providing tools to control access to this shared data. Examples of shared-memory based programming models include the POSIX Pthreads model, the OpenMP model,[38] and the System V interprocess communication model.

In general, the multinode and single node parallel strategies were developed separately based on two different types of parallel architectures. However, it is the evolution of the *node*, specifically the use of multiprocessor "shared-memory" nodes as the building blocks for multinode cluster based systems, which is bringing about a convergence of these methodologies. That is, it is possible to embed the use of shared-memory programming techniques within each node of a cluster based system yet retain the advantages of increased aggregate system resources from a multinode platform. This becomes especially important when examining the roadmap for future generations of computers. The next generation(s) of processors is(are) not expected to dramatically increase in frequency, which traditionally has accounted for 80% of the performance improvements. Rather, the current trend is to add multiple processing "cores" on each processor. This use of multicore processors in multiprocessor nodes further increases the computational density per node and further emphasizes the need to address different parallel strategies for intra- and internode computing and data management within current and future cluster based systems.

The focus of this work is to describe an algorithm for the CCSD(T) method that can utilize both intranode and internode forms of parallelism. Algorithms for parallel CC methods[39−43] have been developed by other groups. These methodologies for the parallelization of CC methods and other correlation methods were divided into two categories: those aimed at shared memory machines (SMPs) and those aimed at distributed memory machines. Early work by Komornicki, Lee, and Rendell[39] described a highly vectorized shared memory algorithm for evaluating the connected triples excitations (T) on the CRAY Y-MP. Vectorized shared-memory CCSD and CCSD(T) algorithms based on AO integrals stored on disk were later implemented by Koch and co-workers.[44,45] These early shared-memory vectorized algorithms primarily used optimized library calls to gain computational speedup (the libraries, not the programs themselves, were multiprocess or multithread based), although some directives to parallelize the loops were employed. Rendell, Lee, and Lindh[40] implemented the first distributed memory CCSD algorithm on an Intel i860 hypercube. In that work, asymptotic speedups were quickly reached due to I/O bottlenecks based on retrieval of the molecular integrals. The authors proposed the use of a "semidirect" method in which atomic integrals evaluated "on demand" could be used to alleviate the I/O bottleneck. Rendell, Guest, and Kendall[41] improved the previous MO-based distributed memory CCSD approach and extended the program to include CCSD(T). Later, Kobayashi and Rendell[42] implemented a "direct" AO-driven CCSD(T) algorithm which avoided the I/O bottlenecks of earlier MO-based distributed memory methods; this development formed the basis for the parallel CCSD(T) module within the NWCHEM package.[27] As another means of avoiding potential I/O bottlenecks, Rendell and Lee proposed[46] that some two-electron integrals can be approximated using the resolution of the identity (RI) technique. RI-based approaches can dramatically reduce the storage requirements needed for CCSD and CCSD(T) calculations, while maintaining $O(N^6)$ and $O(N^7)$ scaling for the computational effort where N is a description of the size of the system being calculated; the number of atomic basis functions is an upper limit to N. MOLPRO[23] also offers a parallel implementation of its coupled cluster methods. Most recently, Janowski and co-workers[43] have presented a parallel algorithm for the CCSD method using the Array Files toolkit.[47]

Another exciting advance in the development of parallel computer codes for high level ab initio quantum chemistry methods is the tensor contraction engine (TCE),[48] a program used for automatic code generation for a general set of high level ab initio methods, including coupled cluster methods. Hirata[49] has shown the utility of the TCE for deriving and implementing many common second-quantized many-electron theories including a variety of coupled cluster methods. The TCE also has the ability to automatically generate *parallel* computer codes. A recent study by Piecuch and co-workers[50] used the TCE to generate a parallel code for the completely renormalized CCSD(T) method[15] which showed that a ten times execution speedup could be achieved using 64 processors. As illustrated by this example, parallel codes generated by the TCE are generally not as efficient as hand-tuned computer codes; however, the major benefits of using the TCE are its ease of use, the avoidance of errors in generating very complex codes, and its general applicability to higher order ab initio methods in which detailed hand tuning and parallelization can be very difficult. The contributions from a number of researchers[51] to the improvement of the generation of highly efficient parallel codes via the TCE program has extraordinary potential and could someday result in automatically generated code that is as good as or better than hand-tuned programs.

The major purpose of this paper is to describe a new parallel CCSD(T) implementation that seeks to find the best balance between the $O(N^7)$ computational cost and the $O(N^4)$ data storage requirements of CCSD(T). The algorithm described here is targeted toward today's basic computer building block: a node with several processor cores and also an appreciable total memory within the node (e.g., $p = 4$ processors and 8 GB of RAM or more). The algorithm also eliminates disk usage while seeking to minimize communications costs. The unique feature of the algorithm presented here is combining the use of both distributed memory (internode) and shared memory (intranode) techniques in a massively parallel (MP) program. Clearly, any MP-CCSD(T) algorithm requires tradeoffs be made in each of these areas, so the proof of the algorithm's viability necessarily must be to demonstrate its ability to do large CCSD(T) computations on realistic hardware, in a reasonable amount of time. The MP-CCSD(T) algorithm described here is an adaptation of the sequential algorithm, previously implemented in GAMESS[22] by Piecuch et al.[52] Because the MP-CCSD(T) method described here is based on the same spin-free equations used by the EOM and renormalized CC methods in GAMESS, the approach to closed shell CCSD-(T) parallelism described here can be extended to the other types of coupled cluster methods in a straightforward manner.

To provide an example of the viability of the MP-CCSD-(T) algorithm on modern cluster based architectures, CCSD-(T) calculations on geometric isomers of water hexamer using the aug-cc-pVTZ basis set[53] are presented. These calculations are important, since there are five low lying isomers of water hexamer (Figure 1), some of which have three-dimensional structures, whose relative energies are very close to each other. Since these are the smallest 3-D water clusters, it is very important to be able to predict the correct energy order for the low-energy isomers with high accuracy. This means that one needs both large basis sets that approach the complete basis set limit, in order to avoid basis set superposition error (BSSE), and a high theoretical level, such

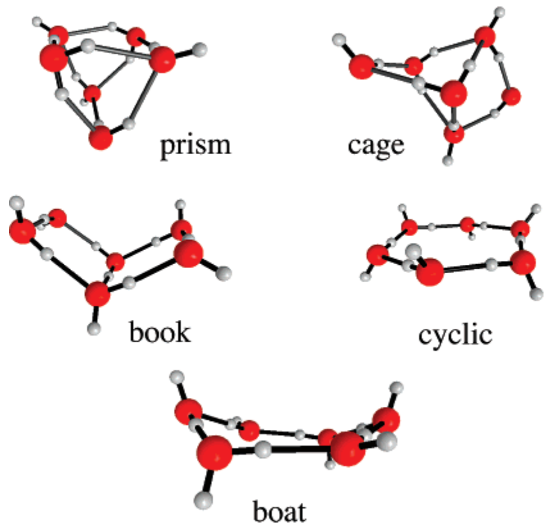


Figure 1. Images of the five geometric isomers used in this study. The geometries correspond to MP2 optimized structures using the DH(d,p) basis set obtained by Day et al.[44]

as CCSD(T). A number of high-level ab initio studies[40−44] have been performed on the water hexamer. In a very thorough and systematic study of the potential energy surface for small water clusters using second-order Moller−Plesset perturbation theory[54] (MP2) and a series of augumented correlation consistent basis sets[53,55] that are systematically improved (aug-cc-pVXZ ranging from X=D,T,Q,5) Xantheas and co-workers[56] have predicted that the prism structure is the global minimum. However, the predicted energy differences among the water hexamer isomers are very small (a range of less than 1.2 kcal/mol for the four isomers studied). Given the known tendency of MP2 to overbind clusters, it is important to employ a more sophisticated level of theory, e.g., CCSD(T), with a sufficiently large basis set such that BSSE approaches zero.[57] The calculations performed herein represent, to the authors' knowledge, the most accurate CCSD(T) calculations on water hexamer to date.

This paper highlights the key features of the MP-CCSD-(T) program and demonstrates that the algorithm is viable on modern cluster based MP platforms. The goal of the MP-CCSD(T) algorithm is to enable high-level CC calculations to provide accurate energies and potential energy surfaces for systems, like water hexamer, that are currently very difficult to achieve. As an illustration of the new method, the CCSD(T)/aug-cc-pVTZ energies for the five low-lying water hexamer isomers are calculated and the performance of the MP-CCSD(T) method is examined. Since the primary focus of the present work is on the MP-CCSD(T) algorithm, the issues of extrapolation to the complete basis set limit and basis set superposition error are deferred to a later paper.

II. CCSD/CCSD(T) Theory

The MP-CCSD(T) method described in this work is an adaptation of the sequential CCSD(T) program previously implemented by Piecuch et al.;[52] therefore, the same notation used in ref 52 is followed here. The letters $i,j,k,l,...$ will be used to denote occupied spatial molecular orbitals, $a,b,c,d,...$ will be used to represent unoccupied (virtual) orbitals, $\mu,\nu,\lambda,\sigma,...$ are used to represent atomic orbital indices or atomic shell indices, and *p*,*q*,*r*,*s*,... are a general set of indices. Details of the CCSD and perturbative (*T*) correction have been discussed in several reviews,[58,59] so only a brief outline is given here.

The CCSD method derives from CC theory in the following manner. Let the exact CC wave function ($|\Psi_{CC}\rangle$) be defined as

$$|\Psi_{CC}\rangle = e^{T}|\Phi\rangle \tag{1}$$

where $|\Phi\rangle$ is the reference wave function (for this work, $|\Phi\rangle$ is the restricted closed-shell Hatree-Fock reference wave function), and *T* is the complete cluster operator containing all possible single (T_1), double (T_2), triple (T_3), etc. excitation operators

$$T = T_1 + T_2 + T_3 + ... \tag{2}$$

The CCSD method results from the truncation of *T* such that only single and double excitation operations are included

$$T \approx T_1 + T_2 \tag{3}$$

Projecting the connected-cluster form of the CCSD equation

$$(H_N e^{T_1+T_2})_C\,|\Phi\rangle = E_{CCSD}\,|\Phi\rangle \tag{4}$$

where H_N is the normal product electronic Hamiltonian ($H - \langle\Phi|H|\Phi\rangle$), onto the set of excited determinants defined by the truncated excitation operator (eq 3) gives rise to a set of coupled nonlinear equations,

$$\langle\Phi_i^a|(H_N e^{T_1+T_2})_C\,|\Phi\rangle = 0 \tag{5}$$

$$\langle\Phi_{ij}^{ab}|(H_N e^{T_1+T_2})_C\,|\Phi\rangle = 0 \tag{6}$$

eqs 5 and 6, which are solved iteratively for the single and double excitations, respectively. In terms of amplitudes (t_i^a, t_{ij}^{ab}), Fock matrix elements ($f_{\;p}^{q}$), and two-electron molecular integrals ($v_{rs}^{pq} = \langle pq|(1/r_{12})|rs\rangle$), respectively, the CCSD amplitude equations (eqs 5 and 6) are given (using the Einstein summation convention). [The Einstein summation convention implies a summation over all possible values of repeated indexes found in the lower or upper positions of a single term. For example, $I_e^a t_i^e = \sum_e I_e^a t_i^e$.]

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

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

In eqs 7 and 8, c_{ij}^{ab} is defined as

$$c_{ij}^{ab} = t_{ij}^{ab} + t_i^a t_j^b \tag{9}$$

The permutation operator *P(ij/ab)* acting on an arbitrary term (*X*) has the following properties

$$P(ij/ab)X_{ij}^{ab} = X_{ij}^{ab} + X_{ji}^{ba} \tag{10}$$

and the general MBPT denominators are used to define D_i^a, D_{ij}^{ab}, and D_{ijk}^{abc} such that

$$D_{ij...}^{ab...} = \epsilon_i - \epsilon_a + \epsilon_j - \epsilon_b ... \tag{11}$$

where

$$\epsilon_p = f_{\;p}^{p} \tag{12}$$

are the diagonal elements of the Fock matrix. The intermediates (*I* and *I′*) of eqs 7 and 8 are

$$I_a^i = f_{\;i}^{a} + 2v_{ae}^{im} t_m^e - v_{ea}^{im} t_m^e \tag{13}$$

$$I_b^a = (1 - \delta_b^a) f_{\;b}^{a} + 2v_{be}^{am} t_m^e - v_{be}^{ma} t_m^e - 2v_{eb}^{mn} c_{mn}^{ea} + v_{be}^{mn} c_{mn}^{ea} - t_m^a f_{\;b}^{m} \tag{14}$$

$$I'^{i}_{j} = (1 - \delta_j^i) f_{\;j}^{i} + 2v_{je}^{im} t_m^e - v_{ej}^{im} t_m^e + v_{ef}^{mi} t_{mj}^{ef} - v_{ef}^{im} t_{mj}^{ef} \tag{15}$$

$$I_j^i = I'^{i}_{j} + I_e^i t_j^e \tag{16}$$

$$I_{kl}^{ij} = v_{kl}^{ij} + v_{ef}^{ij} c_{kl}^{ef} + P(ik/jl) t_e^k v_{el}^{ij} \tag{17}$$

$$I_{jb}^{ia} = v_{jb}^{ia} - \frac{1}{2} v_{eb}^{im} c_{jm}^{ea} - v_{jb}^{im} t_m^a + v_{eb}^{ia} t_j^e \tag{18}$$

$$I_{bj}^{ia} = v_{bj}^{ia} + v_{be}^{im} t_{mj}^{ea} - \frac{1}{2} v_{mb}^{ie} t_{jm}^{ae} - \frac{1}{2} v_{be}^{im} c_{mj}^{ae} + v_{be}^{ia} t_j^e - v_{bj}^{im} t_m^a \tag{19}$$

$$I'^{ab}_{ci} = v_{ci}^{ab} - v_{ci}^{am} t_m^b - t_m^a v_{ci}^{mb} \tag{20}$$

$$I'^{ia}_{jk} = v_{jk}^{ia} + v_{ef}^{ia}\, t_{\;jk}^{ef} + t_j^e v_{be}^{ia} + t_j^e v_{ef}^{ia}\, t_{\;k}^{f} \tag{21}$$

where δ_p^q represents the standard Kronecker delta.

The CCSD correlation energy from the CCSD method is calculated after eqs 7 and 8 are solved iteratively for t_i^a and t_{ij}^{ab} and is given by the following formula

$$\Delta E^{CCSD} = 2f_{\;a}^{i} t_{\;i}^{a} + (2v_{ab}^{ij} - v_{ba}^{ij}) c_{ij}^{ab} \tag{22}$$

Noniterative solutions to the full CCSDT problem using only lower order excitation operators (T_1 and/or T_2) were first developed by Urban and co-workers.[60] These methods eventually led to the CCSD(T) method derived by Raghavachari and co-workers.[5] The (T) of CCSD(T) is an *a posteriori* noniterative correction to the CCSD energy. In a study analyzing a variety of different approximations to the full CCSDT treatment, Scuseria and Lee[6] found the CCSD(T) method to be the most accurate and the most computationally efficient of all the approximate methods examined. In terms of molecular integrals and amplitudes,[52] the correction to the CCSD energy is given by

$$E^{(T)} = \bar{t}_{abc}^{ijk}(2) t_{ijk}^{abc}(2) D_{ijk}^{abc} + \bar{z}_{abc}^{ijk} t_{ijk}^{abc}(2) D_{ijk}^{abc} \tag{23}$$

where an arbitrary $\bar{X}_{abc}^{ijk}$ term is expanded such that

$$\bar{X}_{abc}^{ijk} = \frac{4}{3} X_{abc}^{ijk} - 2X_{acb}^{ijk} + \frac{2}{3} X_{bca}^{ijk} \tag{24}$$

The $t_{ijk}^{abc}(2)$ coefficients are defined in terms of t_i^a and t_{ij}^{ab}

$$D_{ijk}^{abc} t_{ijk}^{abc}(2) = P(ia/jb/kc)[t_{ij}^{ae} v_{ek}^{bc} - t_{im}^{ab} v_{jk}^{mc}] \tag{25}$$

where the permutation operator *P(ia/jb/kc)* expands a quantity containing the (*ia*), (*jb*), and/or (*kc*) pair into a summation of up to six quantities:

$$P(ia/jb/kc)[...]^{abc}_{ijk} = [...]^{abc}_{ijk} + [...]^{acb}_{ikj} + [...]^{cab}_{kij} + [...]^{cba}_{kji} + [...]^{bca}_{jki} + [...]^{bac}_{jik} \quad (26)$$

The second term of eq 23 is the disconnected triples correction to $E^{(T)}$ where

$$z^{ijk}_{abc} \equiv (z^{abc}_{ijk})^* = (t^i_a v^{jk}_{bc} + t^k_c v^{ij}_{ab} + t^j_b v^{ik}_{ac})/D^{abc}_{ijk} \quad (27)$$

and the z^{ijk}_{abc} and z^{abc}_{ijk} are complex conjugates. The final CCSD(T) energy is given by

$$E^{\mathrm{CCSD(T)}} = \langle\Phi|H|\Phi\rangle + \Delta E^{\mathrm{CCSD}} + E^{(T)} \quad (28)$$

A detailed discussion of the individual terms in the CCSD and (T) equations is presented in section 3 of ref 52. The summary presented in eqs 1−28 provides a sufficient background for the following discussion of the implementation of the MP-CCSD(T) method.

III. Parallel Design

There are two primary issues that must be considered in order to perform large-scale CCSD(T) calculations in a massively parallel environment: How can the computational workload be divided among the available parallel processes? How can the large data sets associated with such demanding calculations be stored and utilized efficiently by the available parallel processes?

The amount of computational effort associated with the CCSD and CCSD(T) algorithms scales asymptotically as O(N^6) and O(N^7), respectively. N is a measure of system size and can be broken down more specifically in terms of the number of occupied molecular orbitals (N_o) and the number of unoccupied (virtual) molecular orbitals (N_v). More generally (and more conservatively), one can use the number of one-electron atomic basis functions (N_{bf}). In terms of molecular orbitals, the CCSD and CCSD(T) algorithms scale on the order of their most expensive terms, O($N_o^2N_v^4$) and O($N_o^3N_v^4$), respectively. Each of the terms in the sequential code[52] was parallelized, with specific attention paid to the terms which comprise the *computational bottlenecks.* However, the distribution of the computational work is very closely related to the distribution of the large data sets required by the CCSD(T) method. Therefore, before detailed examples of the manner in which the terms of the CCSD(T) method were parallelized, an examination of data distribution is required.

The second major consideration addresses the storage requirements for large CCSD(T) calculations in a massively parallel environment. As mentioned in section II, the CCSD-(T) equations are written in terms of cluster amplitudes and molecular (or atomic) integrals. The manner in which the integrals and amplitudes are stored on a large parallel computer has a direct effect on how the computational workload can be distributed. Equally important, the choice of how the amplitudes and integrals are stored will directly affect the *storage bottlenecks* of the algorithm.

The MP-CCSD(T) algorithm was designed to address these bottlenecks by first examining the data storage problem and then addressing the parallel work division based on a defined data distribution. In the following discussion, the storage bottlenecks are examined in the scope of the programming model and the available types of storage. Based on these ideas and an outlined storage model, section IV describes how the computational work is divided into internode and intranode components.

A. Parallel Programming Model. The MP-CCSD(T) algorithm introduces and utilizes the third generation of the Distributed Data Interface (DDI) for communication and data storage in a massively parallel environment. The DDI model is a high-level abstraction of the virtual shared-memory model for use in the GAMESS quantum chemistry suite of programs. DDI was designed as a means to provide a consistent set of parallel programming tools for the quantum chemistry code, while maintaining enough generality to be implemented using a variety of existing parallel libraries that offer one-sided message passing, including the following: SHMEM, Global Arrays (GA),[35] MPI,[34] and a native implementation based on point-to-point libraries such as MPI[34] and/or TCP/IP sockets. The DDI model was strongly influenced by the structure and functionality of the Global Arrays (GA) Toolkit; however, to maintain a high degree of portability only a subset of the GA functionality is used within the DDI model.

The first generation of DDI,[37,61] DDI/1, provided a process-based implementation of the distributed-memory programming model in which large arrays could be evenly divided over all available nodes yet remain globally accessible via one-sided message operations. DDI/1 was modeled on the design of the Cray T3E in which the system image of each *node* contained a single processor and some associated system memory. The nodes formed the building blocks of the parallel computer and were connected to other nodes by a high-speed network. DDI/1 is a process-based model, because the data and the computational workload are divided over the parallel processes.

The second generation of DDI,[36] DDI/2, introduced a greater awareness of the memory topology by recognizing that multiple parallel processes could coexist within the same node, i.e., multiple processors in a single node sharing the same local system memory. This shared-memory awareness increases the amount of data that can be considered "local" and can significantly reduce the number of remote communication operations for calculations run using multiprocessor nodes; this was recognized for point-to-point communication in many MPI implementations and also in the one-sided communications for both GA[35] and DDI.[36]

The third generation of DDI, DDI/3, further enhances the shared-memory capabilities of DDI by providing the tools needed for multiprocessor nodes to utilize shared-memory outside of the distributed-memory model. Specifically, DDI/3 provides the ability to create and control access to shared-memory segments as well as the ability to perform point-to-point and collective operations within the node. The shared-memory model in DDI/3 is based on multiple processes using SystemV shared-memory and semaphores for interprocess communication rather than a thread-based model. This maintains the integrity of the former DDI models, whereas a shift to a thread-based model for intranode parallelism would require a radical change to the DDI programming model. DDI/3 provides all the necessary tools for process-based *and* node-based parallelism.

Node-based parallelism differs from process-based parallelism in that the data and the computational work are first divided by node (internode), and then the work assigned to each node is further decomposed and parallelized over the "local" processes within each node. Node-based parallel schemes have the advantage of being able to handle larger replicated data sets when compared to process-based schemes, because shared-memory can be used to store particular quantities *once per node*, rather than *once per process*. The MP-CCSD(T) algorithm described here utilizes both process-based and node-based parallel techniques.

B. Memory. DDI/3 supports three types of memory storage to be used in the MP-CCSD(T) algorithm: replicated, shared, and distributed. Replicated memory is process-based, and the amount of memory needed for data stored in replicated memory scales linearly with the number of processes. Typically, arrays that scale as $O(N^2)$ and some that scale as $O(N^3)$ can be stored in replicated-memory. Shared memory is node-based, and the amount of memory needed for data stored in shared memory scales linearly with the number of nodes. Shared-memory allows for the storage of larger arrays than does replicated-memory, since the arrays are only stored *once per node*. In a shared-memory environment, every process within the node can access and modify the data in shared-memory segments. This feature provides a convenient means of parallelizing the computational work over a shared data set, since each process has direct access to the data in that memory (by physical address). However, allowing multiple processes to have access to shared resources means that special care must be taken to prevent possible race conditions, i.e., situations that occur during parallel execution in which one process seeks to modify data that are concurrently being used by another process. To handle these race conditions, DDI/3 uses SystemV semaphores and collective synchronizations over all intranode processes to control access to shared resources and guarantee data integrity.

Distributed memory is the aggregate of portions of "local" system memory reserved by each process for the storage of distributed data. In the DDI framework, the number of columns of a distributed two-dimensional matrix is divided evenly over the total number of parallel processes; the disjoint sets of columns are mapped in a one-to-one manner onto the set of parallel processes, and the data associated with each set of columns are stored in the memory reserved by each process for distributed memory storage. In contrast to shared memory, access to distributed memory requires calling subroutines from the DDI library. The amount of distributed-memory needed for a given calculation is defined solely by the parameters of the calculation and has *no* dependence on the number of parallel processes used for the calculation. The requirements for distributed memory can in some cases be very large; in those cases, the number of nodes must be chosen to accommodate the required distributed memory.

There are two types of distributed-memory: local and remote. All parallel processes are allowed to modify any element of an array stored in distributed-memory (regardless of physical location); however, due to the communication overhead of accessing remote distributed-memory, the programming strategy seeks to maximize the use of local distributed-memory and minimize the use of remote distributed-memory. In this regard, arrays stored in distributed-memory are not easily rearranged between distributed indexes. For example, when a transpose operation, i.e., the swapping of the rows and columns, is performed on a distributed matrix that is distributed evenly over the number of columns, every parallel process must communicate with all of the other parallel processes. Thus, for very large distributed matrices, this type of operation would require a large amount of communication overhead and would be an impediment to achieving good parallel speedups.

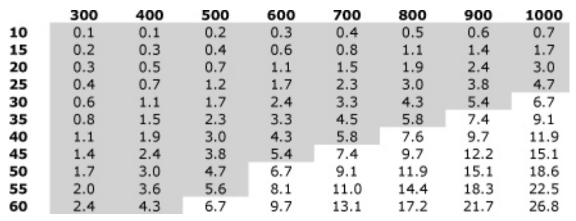
C. Molecular Integral Transformation. The MO integral classes use an "O" to denote an actively correlated occupied MO index and a "V" to denote a virtual MO index. In the present work, a modified version of the distributed-memory "direct" four-index integral transformation[62] previously implemented by Fletcher and co-workers[61] was used to calculate the MO integrals: [OO|OO], [VO|OO], [VV|OO], [VO|VO], and [VV|VO]. The original integral transformation was only able to calculate MO integrals with up to two virtual indexes and was not able to exclude frozen-core MOs from the transformation for a general set of MO integral classes. Modifications were therefore made to allow for the formation of [VV|VO] integrals and to add an option to include or exclude frozen core MOs in the transformation. These modifications maintain the integrity of the original algorithm, i.e., the identical procedures are used; however, the starting indexes and ranges of MO indexes that are transformed were modified.

The formation of the [VV|VO] integrals requires an additional distributed array to store the [NN|VO] integrals, where "*N*" is the total number of basis functions and the entries can be V or O. The same procedure that is used to complete the [VV|OO] integrals from the [NN|OO] set of half-transformed integrals is used to complete the [VV|VO] integrals from the [NN|VO] set of half-transformed integrals. Exclusion of the frozen-core integrals is accomplished using a straightforward modification of the starting index and the range of MO orbitals that are defined as occupied (active). If one wishes to freeze the core molecular orbitals in the coupled cluster calculation, those core molecular orbitals are not correlated, and, therefore, the MO integrals associated with the frozen-core MOs are not required. The option to exclude the frozen-core integrals can result in a significant reduction in computational effort and most importantly a reduction in the distributed-memory requirements for the integral transformation. Of course, for heavier elements such as Au, one must take care in defining those orbitals that are frozen, in order to avoid excluding orbitals that are important in the chemical process of interest.[63]

Table 1. General List of Data Types That Describes What Type of Memory the Quantity Will Be Stored in and How the Quantity Scales as a Function of N_o and N_v

class	type	size	storage
T_1 (t_i^a)	amplitudes	O(N_oN_v)	replicated
T_2 (t_{ij}^{ab})	amplitudes	O($N_o^2N_v^2$)	shared
[OO|OO]	integrals	O(N_o^4)	distributed
[VO|OO]	integrals	O($N_o^3N_v$)	distributed
[VV|OO]	integrals	O($N_o^2N_v^2$)	distributed
[VV|VO]	integrals	O($N_oN_v^3$)	distributed
[VV|VV]	integrals	O(N_v^4)	not stored

Table 2. Maximum Size in Gigabytes (GB) of the Array To Hold the T_2 Amplitudes, the [VV|OO] Integrals, or the [VO|VO] MO Integrals[a]



	300	400	500	600	700	800	900	1000
10	0.1	0.1	0.2	0.3	0.4	0.5	0.6	0.7
15	0.2	0.3	0.4	0.6	0.8	1.1	1.4	1.7
20	0.3	0.5	0.7	1.1	1.5	1.9	2.4	3.0
25	0.4	0.7	1.2	1.7	2.3	3.0	3.8	4.7
30	0.6	1.1	1.7	2.4	3.3	4.3	5.4	6.7
35	0.8	1.5	2.3	3.3	4.5	5.8	7.4	9.1
40	1.1	1.9	3.0	4.3	5.8	7.6	9.7	11.9
45	1.4	2.4	3.8	5.4	7.4	9.7	12.2	15.1
50	1.7	3.0	4.7	6.7	9.1	11.9	15.1	18.6
55	2.0	3.6	5.6	8.1	11.0	14.4	18.3	22.5
60	2.4	4.3	6.7	9.7	13.1	17.2	21.7	26.8

[a] The rows correspond to values of N_o, and the columns correspond to values of N_v. The shaded values correspond to arrays less than or equal to 6 GB.

D. Memory Requirements and Bottlenecks. The scaling of the storage requirements and how the data are stored within the MP-CCSD(T) algorithm are given in Table 1 in terms of the number of actively correlated occupied (N_o) and the number of virtual (N_v) molecular orbitals (MO). In the following discussion, the memory requirements and the potential memory bottlenecks are examined over the range of $10 \le N_o \le 60$ and $300 \le N_v \le 1000$.

For midrange to high-end dedicated supercomputers, the assumption is made that 4−8 GB (GB = 2^{30} bytes) of memory per processor are available. For common 4−8 processor nodes, this means that typically 16−64 GB of "local" system memory per node is generally available. For low-end commodity clusters, these assumptions would not necessarily hold at present; however, it is assumed here that sufficient high-performance computer facilities are available.

Another working assumption is that access to quality disk storage, i.e., "local" multichanneled striped disk arrays on every node, is not generally available. This is a conservative approach to minimize the performance dependence of the algorithm on the quality of the available disk I/O, which can vary greatly from cluster to cluster. In fact, some clusters do not even have local scratch disk storage, and the only available file system may be a remote networked file server or a parallel file system such as Lustre or PVFS2. The performance of the algorithm might be improved if one could assume that quality local disk storage per node is available. In this initial implementation of the MP-CCSD(T) algorithm, only minimal system requirements are assumed.

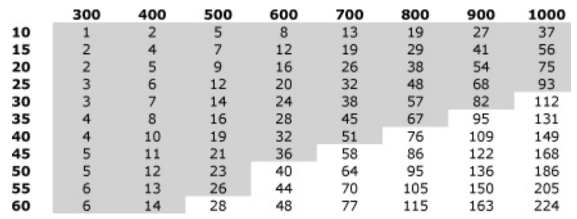
There are two storage bottlenecks in the MP-CCSD(T) algorithm as defined by the choice of data storage (Table 1). These are the storage of the T_2 (t_{ij}^{ab}) in shared-memory and the storage of the [VV|VO] molecular integrals in distributed memory.

The storage of the T_2 (t_{ij}^{ab}) amplitudes in shared-memory is the first of two storage bottlenecks within the MP-CCSD-(T) algorithm. The T_2 amplitudes require $N_o^2N_v^2$ words of shared memory; however, two other intermediates of the same size must also be stored in shared memory. The actual size of the T_2 amplitudes measured in gigabytes (GB) is given in Table 2 (see Table 1 for a summary of all integral and amplitude types). The use of shared memory to store the T_2 amplitudes represents a compromise for the efficient use of the amplitudes, since the T_2 amplitudes are too large (in most cases >1 GB) to be stored in replicated-memory, and these T_2 amplitudes are reordered and manipulated too frequently to be stored in distributed memory. At the limits of $N_o = 60$ and $N_v = 1000$, approximately 27 GB of shared memory per node is required for the T_2 amplitudes. In such cases, the storage of the T_2 amplitudes and the other intermediates is not possible on modern SMP clusters, which, as noted above, typically have 16−64 GB of system memory per node. The present discussion focuses on the implementation for clusters of SMPs; therefore, the practical range of N_o and N_v is defined to be those values for which the size of the T_2 amplitudes is less than 6 GB (the shaded region in Table 2). This practical range of N_o and N_v is defined to overcome the first major storage bottleneck. The same range will be used to examine the sizes for the remaining amplitude and integral classes.

The [VV|OO] and [VO|VO] integrals are similar in size (Table 2) to the T_2 amplitudes, thus over the practical range of N_o and N_v which defines the shared-memory bottleneck of less than 6 GB per array, these quantities are considered small when stored in distributed-memory. Like the T_2 amplitudes, the [VV|OO] and [VO|VO] MO integrals need to be reordered several times throughout the calculation. As mentioned earlier, the reordering of distributed arrays can be very inefficient due to the large amount of communication that is needed. However, unlike the T_2 amplitudes that get updated at the end of every CCSD iteration, the [VV|OO] and [VO|VO] MO integrals are constant for a fixed geometry and basis set. Therefore, instead of reordering the distributed arrays throughout the calculation, two copies of the [VV|OO] integrals and five copies of the [VO|VO] integrals are stored in distributed-memory in the various orders in which they are needed throughout the algorithm. This requires a one-time sorting of the [VV|OO] and [VO|VO] integrals after the integral transformation but prior to the start of the CCSD/CCSD(T) calculation.

The [VV|VO] class of MO integrals is the largest stored quantity in the MP-CCSD(T) algorithm. The distributed-memory needed to store the [VV|VO] integrals represents the second storage bottleneck in the present algorithm. The distributed-memory requirements for the [VV|VO] integrals are given in Table 3. Based on the practical limits of N_o and N_v as governed by the shared-memory bottleneck for the T_2 amplitudes, the largest [VV|VO] distributed-memory arrays can approach ~96 GB. MP-CCSD(T) calculations of this size represent a significant computational challenge. If one employed 128 or more processors for this type of calculation, the storage requirement for the [VV|VO] integrals per

Table 3. Actual Size of the [VV|VO] Integral Class as Stored in Distributed Memory[a]



	300	400	500	600	700	800	900	1000
10	1	2	5	8	13	19	27	37
15	2	4	7	12	19	29	41	56
20	2	5	9	16	26	38	54	75
25	3	6	12	20	32	48	68	93
30	3	7	14	24	38	57	82	112
35	4	8	16	28	45	67	95	131
40	4	10	19	32	51	76	109	149
45	5	11	21	36	58	86	122	168
50	5	12	23	40	64	95	136	186
55	6	13	26	44	70	105	150	205
60	6	14	28	48	77	115	163	224

[a] The values are in gigabytes (GB). The rows correspond to values of N_o, and the columns correspond to values of N_v. The shaded values correspond to those values of N_o and N_v for which the size of the T_2 amplitudes array is less than or equal to 6 GB (Table 2).

Table 4. Size of an N_v^4 Array in Gigabytes (GB)

N_v	300	400	500	600	700	800	900	1000
size (GB)	60	191	466	966	1789	3052	4888	7451

processor would be less than 1 GB. This is easily attained. To decrease the storage requirement of the [VV|VO] integrals, the permutational symmetry of the bra is exploited such that the storage requirement is $[(N_v^2+N_v)\times N_vN_o]/2$ words. When required by the algorithm, the lower triangular $([N_v^2+N_v]/2)$ rows are expanded to a square (N_v^2) set of rows after the data have been received locally. This provides a nearly 2-fold reduction in the storage *and* communication costs, at the cost of a slight increase in computational effort. This tradeoff is logical since the computational resources often cost much less than the memory storage or communication infrastructure.

The [VV|VV] integrals are the largest class of integrals needed for a CCSD calculation; however, due to the $O(N_v^4)$ scaling of the storage requirement for these integrals, the values cannot be practically stored in distributed-memory (Table 1). Only one term in the CCSD equations requires the use of the [VV|VV] integrals. This four-index virtual integral term scales computationally as $O(N_o^2N_v^4)$ and will be referred to here as the *four-virtual term*. An efficient implementation of the four-virtual term is absolutely essential for a CCSD(T) program, since the computational effort required to evaluate the four-virtual term scales as $O(N_v^4)$ with respect to increasing the basis set. Consequently, this is the same rate at which the perturbative triples computation increases using the same metric. By default, most CCSD programs store the [VV|VV] integrals on disk. However, many programs provide the ability to calculate the four-virtual term directly from AO integrals that are calculated "on the fly" rather than stored, thereby avoiding the [VV|VV] integral storage requirement. Methods that avoid storage by calculating quantities "on the fly" are called "direct methods". Table 4 shows the actual storage requirement in gigabytes for the [VV|VV] class of MO integrals. As N_v increases, the memory requirements for the [VV|VV] integrals exceed the distributed-memory capabilities on the vast majority of available computers. The inability to store the [VV|VV] integrals in distributed memory and the general lack of quality disk I/O on large supercomputers led to the implementation of a "direct" four-virtual algorithm that is calculated in parallel from AO integrals. AO driven methods, both direct and disk-based, for CC methods have been studied in the past.[42,44,45] Further details about the direct AO driven four-virtual term are given in section IV.B.

Table 5. Size of N_v^3 Arrays in Megabytes (MB)

N_v	300	400	500	600	700	800	900	1000
size (MB)	206	488	954	1648	2617	3906	5562	7629

Finally, arrays of size N_v^3 are required in both the CCSD and triples correction. For the majority of calculations, N_v^3 arrays are smaller than $N_o^2N_v^2$ arrays; however, as N_v approaches or surpasses N_o^2, these N_v^3 arrays can be similar in size (Table 5) or surpass the size of the T_2 amplitudes array (Table 2). It is for that reason that arrays of size N_v^3 are stored in shared memory and not replicated memory. All other $O(N^3)$ arrays and those of lower order are sufficiently small that they can be stored in the replicated memory of each parallel process.

IV. Parallel Implementation

A. CCSD. Once the Hartree−Fock calculation has converged and the molecular integrals have been calculated and sorted, the CCSD iterations begin. The first part of each CCSD iteration is the evaluation of the direct four-virtual term. The details of this direct calculation are described in section IV.B. As mentioned earlier, the direct evaluation of the four-virtual term eliminates the storage requirements of the [VV|VV] integral class, because the integrals are calculated "on demand" during each iteration. After the four-virtual term has been completed, the MO-based terms (eqs 7 and 8) are evaluated in essentially the same order as in the sequential algorithm. The order in which the terms are evaluated has been designed to reduce the number of floating point operations by maximizing the use of intermediate quantities. The sequential algorithm relies heavily on double-precision general matrix−matrix multiplication, DGEMM, operations for the bulk of the computational effort. [DGEMM is a level 3 BLAS (Basic Linear Algebra Subroutine) library function that performs matrix multiplications.] The node-based parallelization strategy for the DGEMM operations of the CCSD algorithm involves partitioning the DGEMM evenly by the number of nodes. Each node gets one portion of the DGEMM to work on. Then each node divides the DGEMM into equal sized work portions for each process to evaluate.

Another challenging aspect of the parallelization of the CCSD algorithm involves the location of the data, i.e., whether the data for the matrices involved in the DGEMM operation are stored in replicated, shared, or distributed memory. Since the CCSD terms involve contracting integrals and amplitudes via DGEMM operations, and since T_1 and T_2 amplitudes or temporary intermediates of the same size are stored locally on each node (T_1 sized arrays in replicated memory and T_2 sized arrays in shared-memory), the distribution of the MO integrals by node is used in the first partitioning of DGEMM operations. The subsequent intranode partitioning divides the local work among the local processes, where "local" refers to processes within a given node.

For node-based strategies, special care must be taken to ensure the data integrity of shared quantities (both shared and distributed memory arrays); i.e., before a shared quantity can be used, modified, or reordered, a collective synchronization of the processes that have access to the particular quantity must occur. These collective synchronizations, also known as barriers or fences, are points within the program in which all parallel processes of a collective set must enter before any are allowed to continue executing the parallel program. The MP-CCSD(T) algorithm uses the DDI_SYNC subroutine to synchronize the entire set of parallel processes, while the DDI_SMP_SYNC subroutine is used to synchronize all parallel processes that coexist on the same physical node. These collective synchronization routines help safeguard the integrity of shared resources by ensuring that all parallel processes requiring the use of a shared resource have completed a particular task before those processes are allowed to perform new tasks using the same shared resource. An example of this in terms of distributed memory arrays is found in the four-virtual term (section IV.B) where a global synchronization is used to ensure the distributed intermediate ($I_{ij}^{\nu\sigma}$) is complete before the second task of forming I_{ij}^{ab} from $I_{ij}^{\nu\sigma}$ is allowed to begin. However, the most common need for process synchronization occurs when using shared-memory segments within a node. As an example, the evaluation of two CCSD terms that use different orderings of the T_2 amplitudes requires an intranode synchronization to ensure that all local processes have completed the evaluation of the first term and another intranode synchronization to ensure that the entire set of T_2 amplitudes are in the proper order before the evaluation of the second term can begin. This kind of lock-step synchronization can reduce the parallel efficiency of an algorithm if the work between the synchronization points is not evenly balanced.

The following is an example of a node-based algorithm for evaluating the $v_{be}^{am}t_m^e$ component of the I_a^b intermediate (eq 14):

1. Divide n_o by the number of nodes so as to assign each node an equal amount of work.

2. Each node obtains a complete N_v^3 portion of the [VV|VO] integrals from a GET operation based on the index calculated in the previous step, resulting in a 4-index array with dimensions (N_v,N_v,N_v,i) for a given ith index. This array (v_{be}^{ai}) is stored in shared memory. The GET operation is performed only by the master process on each node; therefore, an intranode synchronization is needed before and after this step.

3. Each node performs the permutation of the first and third index, using a routine that allows all the processors on the node to do the permutation in parallel, without overwriting shared memory data. To ensure data integrity, an intranode synchronization is needed after this step is complete.

4. Each node executes a DGEMM (as a $N_v^2 \times N_v$ matrix times a $N_v \times 1$ matrix resulting in a $N_v^2 \times 1$ matrix [$I_a^b = v_{ae}^{bi}t_i^e$ for a fixed i]). This DGEMM is further split among the processes on the node, by dividing N_v^2 (the row dimension of the first matrix) by the number of processors. The actual DGEMM executed by each process consists of a portion of the first matrix times the entire second matrix to yield the entire resultant matrix. In this way, each process works on a different portion of the array. The second matrix and the product matrix are stored in replicated memory on each parallel process.

5. If N_o is greater than the number of nodes, then some (possibly all) nodes will execute steps 2−4 again with a different portion of the [VV|VO] array until the entire matrix multiplication is performed.

6. Local synchronization: A local gather operation is performed to gather the disjoint set of N_v^2 rows of the product matrices into a single N_v^2 matrix on the master process of each node.

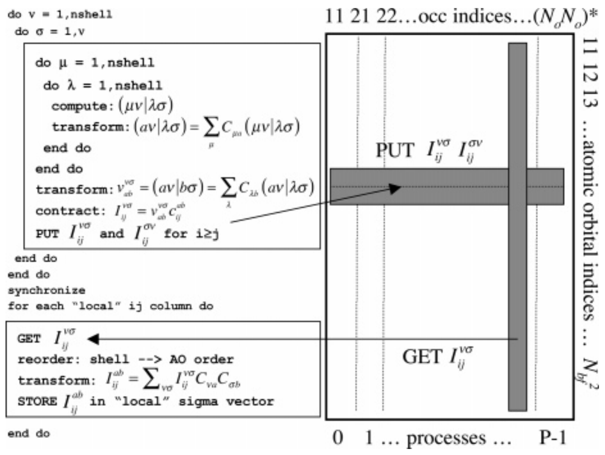
7. The term is completed by a global sum (executed by the master process on each node) over all N_v^2 partial product matrices. A global sum is a form of synchronization.

The remaining terms of the CCSD equations (eqs 7 and 8) have been parallelized using similar techniques to those illustrated in the above example.

In the development of this node-based approach, a similar process-based model was also developed.[64] Depending on the available memory and the size of the calculation, the MP-CCSD(T) parallel algorithm may be evaluated as a process-based algorithm or a node-based algorithm. The more traditional process-based algorithm, which divides the work based on individual processors, may achieve better intranode performance than the node-based model by removing many of the data synchronizations required; however, the process-based algorithm has a larger memory requirement due to the necessity of more replicated temporary memory, and this significantly limits the size of a molecular problem that can be addressed. Therefore, although both process-based and node-based algorithms have been developed and implemented,[64] only the node-based algorithm is discussed here, as a primary focus of this discussion to extend the size and complexity of molecular species that can be studied with CCSD(T) methods.

B. "Direct" AO-Driven Four-Virtual Term. The AO "direct" four-virtual term is a distributed-memory algorithm that makes use of both node-based and process-based parallel techniques. The parallel programming techniques used to implement the four-virtual term of the MP-CCSD(T) algorithm were inspired by both the implementation by Fletcher and co-workers[61] of the direct four-index integral transformation[62] and the direct CCSD algorithm of Kobayashi et al.[42] Diagrammatic representations of these algorithms are illustrated in Figure 2 (the new MP-CCSD(T) algorithm presented here), Figure 3 (the [VV|OO] integral class of the integral transformation of Fletcher et al.[61]), and Figure 4 (four-virtual term of Kobayashi et al.[42]). The four-virtual terms of each CCSD algorithm contracts AO integrals and amplitudes to calculate the contribution of the four-virtual term to each CCSD iteration.

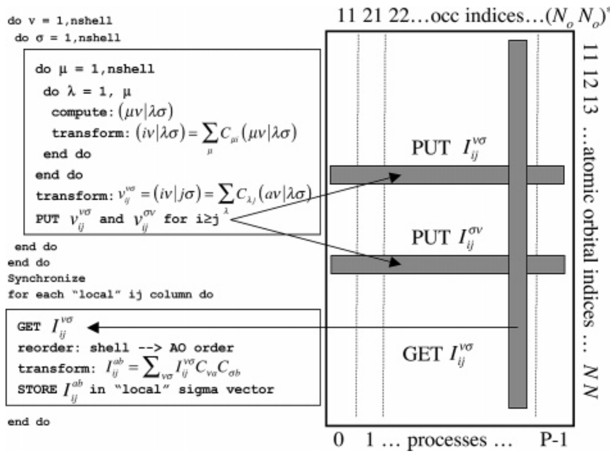
The four-virtual term of the MP-CCSD(T) algorithm (Figure 2) "directly" calculates full sets of two virtual-indexed half-transformed MO integrals ($v_{ab}^{\nu\sigma}$) for the specific shell indices ν and σ. The half-transformed integrals are then contracted against the c_{ij}^{ab} amplitudes ($c_{ij}^{ab} = t_{ij}^{ab} + t_i^a t_j^b$) to form a partial contribution to the set of half-transformed intermediates ($I_{ij}^{\nu\sigma}$), which are complete for a given set of atomic shells ν and σ (Figure 2) corresponding



```
do v = 1,nshell
 do σ = 1,v
  do μ = 1,nshell
   do λ = 1,nshell
    compute: $(\mu\nu|\lambda\sigma)$
    transform: $(a\nu|\lambda\sigma) = \sum_{\mu} C_{\mu a}(\mu\nu|\lambda\sigma)$
   end do
  end do
  transform: $v_{ab}^{\nu\sigma} = (a\nu|b\sigma) = \sum_{\lambda} C_{\lambda b}(a\nu|\lambda\sigma)$
  contract: $I_{ij}^{\nu\sigma} = v_{ab}^{\nu\sigma} c_{ij}^{ab}$
  PUT $I_{ij}^{\nu\sigma}$ and $I_{ij}^{\sigma\nu}$ for i≥j
 end do
end do
synchronize
for each "local" ij column do
  GET $I_{ij}^{\nu\sigma}$
  reorder: shell --> AO order
  transform: $I_{ij}^{ab} = \sum_{\nu\sigma} I_{ij}^{\nu\sigma} C_{\nu a} C_{\sigma b}$
  STORE $I_{ij}^{ab}$ in "local" sigma vector
end do
```

11 21 22…occ indices…(N_oN_o)*
11 12 13 …atomic orbital indices … N_{bf}^2
PUT $I_{ij}^{\nu\sigma}$ $I_{ij}^{\sigma\nu}$
GET $I_{ij}^{\nu\sigma}$
0 1 … processes … P-1

Figure 2. A diagrammatic description of the four-virtual term in the MP-CCSD(T) algorithm. The left-hand portion of the diagram is pseudocode, while the right-hand portion illustrates a distributed array. The columns of the distributed array correspond to two occupied indexes, where the total number of columns is (N_oN_o)*. (N_oN_o)* refers to the lower triangular portion (including diagonal elements) of an N_o^2 matrix. The number of rows in the distributed matrix is N_{bf}^2. The columns are distributed evenly over the total number of parallel processes. The boxed portions of the pseudocode represent loadbalanced parallel tasks. The first half of the pseudocode forms the half-transformed intermediate ($I_{ij}^{\nu\sigma}$) in distributed memory. A global synchronization is used to ensure $I_{ij}^{\nu\sigma}$ is complete before the second parallel task begins. The second parallel task transforms $I_{ij}^{\nu\sigma}$ into I_{ij}^{ab} for all "local" *i-j* columns.



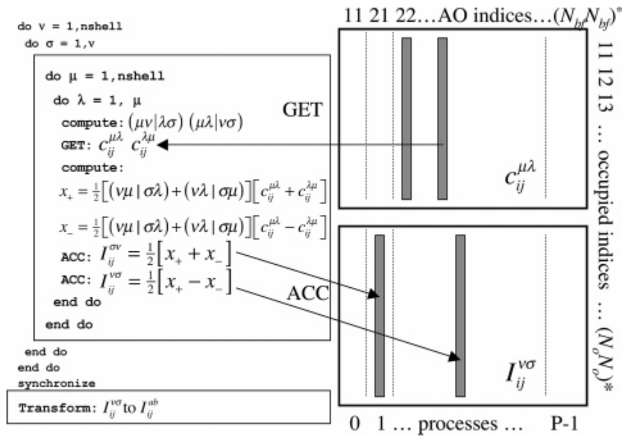
```
do v = 1,nshell
 do σ = 1,nshell
  do μ = 1,nshell
   do λ = 1, μ
    compute: $(\mu\nu|\lambda\sigma)$
    transform: $(i\nu|\lambda\sigma) = \sum_{\mu} C_{\mu i}(\mu\nu|\lambda\sigma)$
   end do
  end do
  transform: $v_{ij}^{\nu\sigma} = (i\nu|j\sigma) = \sum_{\lambda} C_{\lambda j}(a\nu|\lambda\sigma)$
  PUT $v_{ij}^{\nu\sigma}$ and $v_{ij}^{\sigma\nu}$ for i≥j
 end do
end do
Synchronize
for each "local" ij column do
  GET $I_{ij}^{\nu\sigma}$
  reorder: shell --> AO order
  transform: $I_{ij}^{ab} = \sum_{\nu\sigma} I_{ij}^{\nu\sigma} C_{\nu a} C_{\sigma b}$
  STORE $I_{ij}^{ab}$ in "local" sigma vector
end do
```

Figure 3. A diagrammatic description of the Fletcher[50] four-index integral transformation for the [VV|OO] integral class. The left-hand portion of the diagram is pseudocode, while the right-hand portion illustrates a distributed array. The columns of the distributed array correspond to two occupied indexes, where the total number of columns is (N_oN_o)*. (N_oN_o)* refers to the lower triangular portion (including diagonal elements) of an N_o^2 matrix. The number of rows in the distributed matrix is N_{bf}^2. The columns are distributed evenly over the total number of parallel processes. The boxed portions of the pseudocode represent load-balanced parallel tasks. The first half of the pseudocode forms half-transformed integrals over two occupied indexes for a given set of two AO shell indexes. The half-transformed integrals are stored in the distributed array. A global synchronization is used to ensure the first task is complete before the second parallel task begins. The second parallel task transforms the final two AO indexes into virtual MO indexes.

to the parallel task (a given ν-σ pair). For each parallel task the half-transformed intermediates $I_{ij}^{\nu\sigma}$ and $I_{ij}^{\sigma\nu}$ for $i \geq j$ are stored in distributed memory. After the first set of parallel tasks is complete, the full set of half-transformed intermediates $I_{ij}^{\nu\sigma}$ (for $i \geq j$) is stored in distributed memory. To finalize the contributions of the four-virtual CCSD term, the two remaining AO indices are transformed to the virtual MO space.

The four-virtual term gains potential performance advantages over the integral transformation on which it is modeled in two ways: an improved computation vs communication ratio and a reduction in the total number of communication calls. The first parallel task of the four-virtual term (Figure 2) evaluates a larger number of AO integrals and then forms a larger set of half-transformed integrals than the integral transformation in the Fletcher algorithm (Figure 3). In addition, the extra $O(N_o^2N_v^2)$ contraction step makes the first parallel task of the four-virtual term of the MP-CCSD(T) algorithm significantly more computationally challenging than the first parallel task of the integral transformation. However, both methods share a similar communication profile, which places all of the communication at the end of the parallel task. In fact, the PUT operations performed for the four-virtual term in the MP-CCSD(T) algorithm communicate and store the same amount of data in distributed-memory as the integral transformation does in the formation of the [NN|OO] set of half-transformed integrals. The PUT operations in the four-virtual term of the MP-CCSD(T) algorithm actually gain a slight edge over the PUT operations of the integral transformation in that in the former, only *one* PUT operation is performed at the end of the first parallel task of the four-virtual term. In contrast, potentially two PUT operations are performed in the integral transformation, except for a single PUT operation when $\nu = \sigma$. Due to the larger *computational* profile of the four-virtual term and a *communication* profile that is similar to the integral transformation, the four-virtual term of the MP-CCSD(T) algorithm is expected to be as good or better in terms of computational efficiency when compared to the integral transformation. The latter has previously been shown to be highly efficient up to 512 processors.[65]

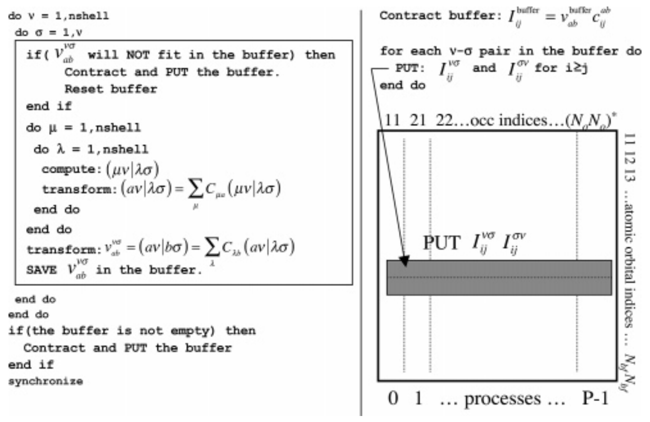
Both distributed-memory CCSD algorithms examined herein form the half-transformed intermediate $I_{ij}^{\nu\sigma}$ of the four-virtual term in distributed memory (Figures 2 and 4). The major difference between the two algorithms is the communication profile. In the four-virtual term of Kobayashi et al.,[42] the communication calls (GET and ACC) are performed on the inner most nested loop (Figure 4). This type of algorithm was shown to be very successful on the Cray T3E. However, the Cray T3E is very different from modern HPC platforms in that the performance of modern processors has increased by more than an order of magnitude, while the performance of the communication networks have at best doubled or tripled since the benchmarks on the T3E. Therefore the communication heavy inner loop [O(N^4)



```
do ν = 1,nshell
 do σ = 1,ν
  do μ = 1,nshell
   do λ = 1, μ
    compute: $(\mu\nu|\lambda\sigma)$ $(\mu\lambda|\nu\sigma)$
    GET: $c_{ij}^{\mu\lambda}$ $c_{ij}^{\lambda\mu}$
    compute:
    $x_+ = \frac{1}{2}\left[(\nu\mu|\sigma\lambda)+(\nu\lambda|\sigma\mu)\right]\left[c_{ij}^{\mu\lambda}+c_{ij}^{\lambda\mu}\right]$
    $x_- = \frac{1}{2}\left[(\nu\mu|\sigma\lambda)+(\nu\lambda|\sigma\mu)\right]\left[c_{ij}^{\mu\lambda}-c_{ij}^{\lambda\mu}\right]$
    ACC: $I_{ij}^{\sigma\nu} = \frac{1}{2}\left[x_+ + x_-\right]$
    ACC: $I_{ij}^{\nu\sigma} = \frac{1}{2}\left[x_+ - x_-\right]$
   end do
  end do
 end do
end do
synchronize
Transform: $I_{ij}^{\nu\sigma}$ to $I_{ij}^{ab}$
```

11 21 22…AO indices…$(N_{bf}N_{bf})^*$
GET
$c_{ij}^{\mu\lambda}$
11 12 13 … occupied indices … $(N_oN_o)^*$
ACC
$I_{ij}^{\nu\sigma}$
0 1 … processes … P-1

Figure 4. A diagrammatic description of the four-virtual term of Kobayashi et al.[33] The left-hand portion of the diagram is pseudocode, while the right-hand portion illustrates the distributed arrays. The columns of the distributed arrays correspond to two AO indexes where the total number of columns is $(N_{bf}N_{bf})^*$. $(N_{bf}N_{bf})^*$ refers to the lower triangular portion (including diagonal elements) of an N_{bf}^2 matrix. The number of rows in the distributed matrix is $(N_oN_o)^*$ corresponding to the lower triangular portion of an N_o^2 matrix. The columns are distributed evenly over the total number of parallel processes. The boxed portions of the pseudocode represent load-balanced parallel tasks. The first half of the pseudocode forms the half-transformed intermediate ($I_{ij}^{\nu\sigma}$) in distributed memory. A global synchronization is used to ensure $I_{ij}^{\nu\sigma}$ is complete before the second parallel task begins. The second parallel task transforms $I_{ij}^{\nu\sigma}$ into I_{ij}^{ab} for all "local" *i-j* columns.



```
do ν = 1,nshell
 do σ = 1,ν
  if( $V_{ab}^{\nu\sigma}$ will NOT fit in the buffer) then
       Contract and PUT the buffer.
       Reset buffer
  end if
  do μ = 1,nshell
   do λ = 1,nshell
    compute: $(\mu\nu|\lambda\sigma)$
    transform: $(a\nu|\lambda\sigma) = \sum_{\mu} C_{\mu a}(\mu\nu|\lambda\sigma)$
   end do
  end do
  transform: $v_{ab}^{\nu\sigma} = (a\nu|b\sigma) = \sum_{\lambda} C_{\lambda b}(a\nu|\lambda\sigma)$
  SAVE $V_{ab}^{\nu\sigma}$ in the buffer.
 end do
end do
if(the buffer is not empty) then
  Contract and PUT the buffer
end if
synchronize

Contract buffer: $I_{ij}^{\text{buffer}} = v_{ab}^{\text{buffer}} c_{ij}^{ab}$

for each ν-σ pair in the buffer do
  PUT: $I_{ij}^{\nu\sigma}$ and $I_{ij}^{\sigma\nu}$ for i≥j
end do
```

11 21 22…occ indices…$(N_oN_o)^*$
11 12 13 …atomic orbital indices … $N_{bf}N_{bf}$
PUT $I_{ij}^{\nu\sigma}$ $I_{ij}^{\sigma\nu}$
0 1 … processes … P-1

Figure 5. A more detailed overview of the first parallel task in Figure 2 describing the use of a temporary buffer to store half-transformed integrals. The left-hand portion of the figure is the pseudocode describing how the buffer is filled. The right-hand portion describes the "Contract and PUT" operation on the buffer. The description of the distributed array is the same as in Figure 2.

communication calls] is less favorable on modern MP platforms due to this growing discrepancy of the communication network compared to the available computational power. The main benefit of the MP-CCSD(T) routine is that the communication operations are performed at the end of the parallel task making the number of communication calls scale as O(N^2) (Figure 2). The GET operation of the Kobayashi et al.[42] method is avoided completely by the storage of the c_{ij}^{ab} amplitudes in shared-memory once on every node. The ACC operation of the Kobayashi et al.[42] method is replaced by a less expensive PUT operation, since the set of $I_{ij}^{\nu\sigma}$ is complete for each set of ν and σ.

The diagrammatic description of the four-virtual term in the MP-CCSD(T) method (Figure 2) is a general description of the algorithm. The actual algorithm as programmed in GAMESS incorporates an extra step to further optimize the first parallel task (see Figure 5). To maximize the efficiency in the contraction step, a local buffer is used to store multiple sets of half-transformed integrals prior to the DGEMM operation. Without the use of the buffer, the size of the DGEMM operation is a function of the size of the basis set shells ν and σ. When ν and σ are s-shells, the DGEMM contraction step reduces to a less than optimal DGEMV (matrix times vector) operation. By locally buffering sets of half-transformed integrals (Figure 5), the efficiency of the DGEMM operation is increased because larger more efficient DGEMM operations are calculated rather than multiple sets of smaller less efficient DGEMV operations. The PUT operation for each set of ν and σ is then performed for each ν-σ pair in the contracted buffer.

C. Triples Correction, MP-(T). The (T) portion of the MP-CCSD(T) algorithm is more straightforward to parallelize than the CCSD component. It consists of three nested loops, each of size n_i,n_j,n_k with $i \geq j \geq k$, where *i*, *j*, and *k* are actively occupied indexes. Within each loop, 36 DGEMM calls are made, the largest of which scales computationally as O(N_v^4) and corresponds to DGEMM operation where a $N_v \times N_v$ matrix is multiplied by a $N_v \times N_v^2$ matrix. One feature of the (T) algorithm is that the loop iterations can be performed independently of each other, thus the algorithm can be easily partitioned into unique parallel tasks. The node-based (T) algorithm partitions these independent tasks in terms of sets based on unique values of *i*, *j*, and *k* (occupied indexes), where each task is evaluated on a node. Two N_v^3 temporary arrays are stored one time per node in shared memory. Similar to the parallelization scheme of the MO-based MP-CCSD algorithm, when a computationally intensive routine (such as a permutation or DGEMM) is encountered, the work is partitioned equally among the intranode processes, with strict control maintained to avoid overwriting shared memory array locations by multiple processors.

The intranode scaling of the MP-(T) algorithm is expected to exhibit similar trends to those of the MP-CCSD algorithm, since the lock-step synchronization needed between the intranode processes within the node-based tasks are similar. However, due to the larger amount of computational effort per parallel task, the MP-(T) algorithm is expected to perform better.

In general, the MP-(T) algorithm has a large number of independent tasks that are similar to the four-virtual algorithm; however, unlike the four-virtual algorithm, the MP-(T) algorithm does not evaluate the integrals it requires on demand. Rather, it fetches them via GET operations. This aspect of the MP-(T) algorithm increases the communication overhead of the algorithm; however, the O(N_v^4) effort within each parallel task easily compensates to allow for a

Table 6. Total Energies (Hartree), Binding Energies (kcal/mol), and Relative Binding Energies (kcal/mol) Using the aug-cc-pVTZ Basis Set on the MP2/DH(d,p) Optimized Structures of Day et al.[a]

	CCSD(T)	CCSD	MP2	MP2*
		Total Energies (Hartree)		
prism	−458.13045167	−458.07282232	−458.05015535	−458.05035804
cage	−458.13001003	−458.07248668	−458.05001662	−458.05017138
book	−458.12851572	−458.07140532	−458.04884875	−458.04960143
cyclic	−458.12704114	−458.07054907	−458.04769324	−458.04785303
boat	−458.12514762	−458.06876898	−458.04579806	
		Binding Energies (kcal/mol)		
prism	−48.1	−44.6	−47.9	−47.9
cage	−47.8	−44.4	−47.8	−47.8
book	−46.9	−43.7	−47.1	−47.5
cyclic	−46.0	−43.2	−46.4	−46.4
boat	−44.8	−42.1	−45.2	n/a
		Relative Binding Energies with Respect to the Prism Isomer (kcal/mol)		
prism	0.0	0.0	0.0	0.0
cage	0.3	0.2	0.1	0.1
book	1.2	0.9	0.8	0.4
cyclic	2.1	1.4	1.5	1.5
boat	3.3	2.5	2.7	n/a

[a] The binding energies represent the energy difference between the water hexamer isomer and six isolated water molecules. MP2* represents the MP2/aug-cc-pVTZ calculations from Xantheas et al. who did not examine the boat isomer.

favorable computational vs communication ratio. Therefore, good internode scalability is expected from the MP-(T) routine.

V. Computational Details

The starting set of geometries for the five water hexamer isomers (prism, cage, book, ring, and boat) was obtained from Day et al.[66] (Figure 1**)**. In that work, the geometries of water hexamer were optimized with second-order Moller−Plesset perturbation theory (MP2)[54] using the double-ζ Dunning-Hay[67] [DH(d,p)] basis set. In the present work, single-point CCSD(T) energies were calculated at each previously optimized structure using the following one-electron basis sets: aug-cc-pVTZ[53] and aug′-cc-pVTZ, where aug′-cc-pVTZ is a mixed basis set that uses aug-cc-pVTZ on the oxygen atoms and cc-pVTZ[55] on the hydrogen atoms. The MP-CCSD(T) method in GAMESS was used for all CCSD(T) calculations. A cluster of three IBM Power4 compute nodes each containing eight 1.7 GHz Power4 processors and 32 GB of memory connected by TCP/IP over an InfiniBand network was used to perform the MP-CCSD(T) calculations.

To evaluate the performance of the MP-CCSD(T) algorithm, a series of CCSD(T)/aug′-cc-pVTZ calculations were performed on the MP2/DH(d,p) optimized prism isomer, and the parallel execution times were measured. To test intranode scalability, a set of CCSD(T)/aug′-cc-pVTZ//MP2/DH(d,p) energies were calculated using a single node; the number of parallel processes was varied from 1 to 8 in powers of 2. Internode scalability measures the changes in parallel runtime as the number of nodes is increased, while the number of parallel processes per node (1, 2, 4, or 8) is fixed. In terms of N_o, N_v and the number of Cartesian basis functions N_{bf}), the size of the aug-cc-pVTZ calculation is $N_o = 24$, $N_v = 516$, and $N_{bf} = 630$. The size of the aug′-cc-pVTZ calculation is $N_o = 24$, $N_v = 408$, and $N_{bf} = 510$.

VI. Discussion

Water Hexamer. Calculations performed on the isomers of the water hexamer were used to test the MP-CCSD(T) algorithm. In the first step of what will be a more extensive study of water clusters, CCSD(T) single point energies using the aug-cc-pVTZ and the aug′-cc-pVTZ basis sets were calculated at the MP2 optimized geometries of Day et al.[66] The absolute energies, binding energies per H_2O, and relative binding energies are given in Table 6 for calculations using the aug-cc-pVTZ basis set and for calculations using the aug′-cc-pVTZ basis set in Table 7. These calculations represent, to the authors' knowledge, the largest CCSD(T) calculations performed on water hexamers to date.

The MP2/DH(d,p) geometries of Day et al.[66] used in this study may not be as accurate as the MP2/aug-cc-pVTZ geometries of Xantheas et al.;[56] however, the differences in the binding energies for the two sets of geometries (Table 6) are very small: < 0.1 kcal/mol for the prism, cage, and cyclic isomers and 0.4 kcal/mol for the book isomer. The latter suggests that calculations reported below based on MP2/DH(d,p) geometries for the book isomer may not be as accurate as those for the prism, cage, and cyclic isomers. Xantheas and co-workers did not examine the boat structure.

A main point of interest in this study is the difference between the CCSD(T) and MP2 methods. Column 1 of Table 8 shows the difference in CCSD(T) vs MP2 relative binding energies; positive values indicate an increase in the energy difference between an isomer and the lowest energy prism isomer, i.e., the value in which the prism isomer is stabilized by the CCSD(T) method. In general, CCSD(T) and MP2 predict very similar binding energies. CCSD(T) moderately stabilizes the prism structure with respect to the cage and other higher energy isomers. The prism isomer is stabilized by 0.2 kcal/mol over the next lowest-energy cage isomer. For higher energy isomers, the difference in relative binding

Table 7. Total Energies (Hartree), Binding Energies (kcal/mol), and Relative Binding Energies (kcal/mol) Using the aug′-cc-pVTZ Basis Set on the MP2/DH(d,p) Optimized Structures of Day et al.[a]

	CCSD(T)	CCSD	MP2
	Total Energies (Hartree)		
prism	−458.12255430	−458.06558835	−458.04247161
cage	−458.12223107	−458.06536922	−458.04244064
book	−458.12086894	−458.06440785	−458.04140132
cyclic	−458.11967703	−458.06381033	−458.04052628
boat	−458.11779063	−458.06203960	−458.03863436
	Binding Energies (kcal/mol)		
prism	−46.6	−43.2	−46.6
cage	−46.4	−43.1	−46.6
book	−45.6	−42.5	−45.9
cyclic	−44.8	−42.1	−45.4
boat	−43.6	−41.0	−44.2
	Relative Binding Energies with Respect to the Prism Isomer (kcal/mol)		
prism	0.0	0.0	0.0
cage	0.2	0.1	0.0
book	1.0	0.7	0.6
cyclic	1.8	1.1	1.2
boat	3.0	2.2	2.4

[a] The binding energies represent the energy difference between the water hexamer isomer and six isolated water molecules.

energies is larger: 0.4 kcal/mol for the book isomer and 0.6 kcal/mol for the cyclic and boat isomers. While the differences in relative binding energies between CCSD(T) and MP2 (0.2−0.6 kcal/mol) are modest when the aug-cc-pVTZ basis set is used, it is unclear how basis set improvements will affect these energy differences.

Another interesting issue is the accuracy of the CCSD method with respect to CCSD(T) and MP2. The CCSD(T) and MP2 binding energies agree to within ∼0.5 kcal/mol for both the aug- and aug′-cc-pVTZ basis sets (Tables 6 and 7). However, the CCSD binding energies differ from the CCSD(T) binding energies by 2.7−3.5 kcal/mol (Tables 6 and 7). Assuming that CCSD(T) provides the most accurate binding energies, these calculations suggest that the MP2 method can more accurately predict the binding energies than the CCSD method. Kim et al.[68] reported such a difference between CCSD and MP2 for cyclic water hexamer. This is surprising, since the CCSD method is often considered to be more reliable than MP2.

Table 8 describes in more detail how the prism isomer is stabilized by the CCSD(T) method based on differences between CCSD(T) and CCSD (column 2) and differences between CCSD and MP2 (column 3). The triples correction to the CCSD energy (column 2, Table 8) plays an increasingly larger role in stabilizing the prism structure relative to higher energy isomers. The difference between CCSD and MP2 (column 3, Table 8) stabilizes the prism structure over the cage and book structures but decreases the stability of the prism structure relative to the cyclic and boat structures. The effects of the triples approach 1 kcal/mol and should not be overlooked, especially for larger water clusters.

The two basis sets employed here exhibit very similar trends in the differences of relative binding energies for all

Table 8. Difference in Relative Binding Energies between the CCSD(T), CCSD, and MP2 Methods with Respect to the Basis Set Measured in kcal/mol[a]

	CCSD(T)-MP2	(T)-CCSD	CCSD-MP2
		aug-cc-pVTZ	
prism	0.00	0.00	0.00
cage	0.19	0.07	0.12
book	0.39	0.33	0.07
cyclic	0.60	0.71	−0.12
boat	0.59	0.78	−0.19
		aug′-cc-pVTZ	
prism	0.00	0.00	0.00
cage	0.18	0.07	0.12
book	0.39	0.32	0.07
cyclic	0.58	0.69	−0.10
boat	0.58	0.76	−0.18
		difference	
prism	0.00	0.00	0.00
cage	0.01	0.00	0.01
book	0.01	0.01	0.00
cyclic	0.01	0.02	−0.01
boat	0.01	0.02	−0.01

[a] The first column [CCSD(T)-MP2] shows how the relative binding energies differ between the CCSD(T) and the MP2 method. The second column [CCSD(T)-CCSD] shows the effect of the triples correction on the relative binding energies. The last column [CCSD-MP2] shows the differences between CCSD and MP2 on the relative binding energies. The section subtitled “difference” subtracts the values from the aug-cc-pVTZ set from the corresponding aug′-cc-pVTZ set.

methods. Csonka and co-workers[69] have suggested that including diffuse functions in the oxygen atom basis set is important. In the present work, the aug′-cc-pVTZ basis set only includes diffuse functions on the oxygen atoms. The omitted hydrogen diffuse functions in the aug′-cc-pVTZ basis set were found to increase the binding energies of the water clusters by 1.0−1.4 kcal/mol (Table 6 and Table 7); therefore the diffuse functions on the hydrogen atoms do seem to be important for calculating the *absolute* binding energies. However, the *relative* binding energies (Table 8) predicted by the aug- and aug′-cc-pVTZ basis sets are very similar. For example, column 1 of Table 8 describes the differences in relative binding energies between CCSD(T) and MP2. These values are virtually identical for each isomer for both basis sets. This consistency in the differences between CCSD(T) and MP2 for the aug- and aug′-cc-pVTZ basis sets suggests that the CCSD(T)/aug-cc-pVTZ binding energies can be accurately estimated using computationally less intensive CCSD(T)/aug′-cc-pVTZ and MP2/aug-cc-pVTZ calculations. As illustrated in Table 9, the following additive scheme,

$$\text{CCSD(T)/aug} = \text{CCSD(T)/aug}' + [\text{MP2/aug} - \text{MP2/aug}'] \quad (29)$$

where the -cc-pVTZ extension to the basis set is implied, estimates the actual CCSD(T)/aug-cc-pVTZ binding energies to within less than 0.2 kcal/mol. A future study will examine

Table 9. Estimated CCSD(T)/aug-cc-pVTZ Binding Energies (kcal/mol) Using Eq 29 Compared to the Actual CCSD(T)/aug-cc-pVTZ at the MP2/DH(d,p) Optimized Geometries[a]

	prism	cage	book	cyclic	boat
est. CCSD(T)/ aug-cc-pVTZ	−47.9	−47.6	−46.7	−45.7	−44.6
actual CCSD(T)/ aug-cc-pVTZ	−48.1	−47.8	−46.9	−45.9	−44.8
error	0.2	0.2	0.2	0.2	0.2

[a] The differences were rounded up.

Table 10. Parallel Speedup (*S*) and Parallel Efficiency (*E*) for the MP-CCSD(T) Algorithm as a Function of the Number of Processors per Node (PPN) and the Number of Nodes for Calculations Performed on the Prism Isomer Using the aug′-cc-pVTZ Basis Set[a]

processes per node	1		2		4		8	
	S	*E* (%)	*S*	*E* (%)	*S*	*E* (%)	*S*	*E* (%)
1 Node								
CCSD-AO	1.00	100	1.90	95	3.70	92	6.18	77
CCSD-MO	1.00	100	1.87	93	3.11	78	4.21	53
CCSD-total	1.00	100	1.86	93	3.58	89	5.68	71
triples correction (T)	1.00	100	1.78	89	2.59	65	4.06	51
2 Nodes								
CCSD-AO	2.00	100	3.76	94	7.43	93	12.31	77
CCSD-MO	1.38	69	2.46	62	4.10	51	6.21	39
CCSD-total	1.88	94	3.34	84	6.53	82	9.56	60
triples correction (T)	1.94	97	3.38	85	4.73	59	7.13	45
3 Nodes								
CCSD-AO	3.00	100	5.85	97	11.07	92	18.48	77
CCSD-MO	1.68	56	2.96	49	4.56	38	6.91	29
CCSD-total	2.55	85	4.80	80	8.28	69	14.57	61
triples correction (T)	2.95	98	5.24	87	7.63	64	11.82	49

[a] CCSD-AO represents the AO-driven four virtual term of the MP-CCSD algorithm; CCSD-MO represents all the other MO-based terms of the MP-CCSD algorithm. CCSD-total is the overall scalability for the MP-CCSD algorithm. The speedup and efficiency is also given for the triples correction. Intranode trends are observed across rows, while internode trends are observed down the columns. The benchmark calculations are based on MP-CCSD(T) calculations of the water hexamer (prism isomer) with $N_o = 24$, $N_v = 408$, and $N_{bf} = 510$ run on nodes containing a total of 8 processors.

the extrapolation of the CCSD(T) binding energies of water hexamer isomers to the complete basis set limit (CBS).

Parallel Performance. The speedup and efficiency values for the four virtual term (CCSD-AO) and the remaining MO terms (CCSD-MO) from the MP-CCSD method on the benchmark calculation run on the IBM Power4 platform are given in Table 10. Speedup is defined as the ratio of the measured execution time to the execution time on a single processor; efficiency is the ratio of the measured speedup compared to the ideal speedup.

The intranode scalability of the MP-CCSD method was measured by the speedup and efficiency of the benchmark calculation as the number of processors within a single node was increased. The intranode scalability of the AO driven four-virtual term (CCSD-AO) is better than 90% of ideal over two and four processors within one node; however, the efficiency drops to approximately 77% when all eight processors within the node are used (Table 10). The drop in performance when using all eight processors with a single node is likely due to memory bandwidth limitations; i.e. all eight processors within the node were accessing and utilizing the same local system memory. The scalability of the MO based terms of the MP-CCSD algorithm is approximately 93%, 77%, and 52% efficient when run on 2, 4, and 8 processors, respectively, within the same node (Table 10). The intranode scalability of the MO based MP-CCSD terms suffers due to the high degree of synchronization needed between local processes; the lock-step manner in which the terms are calculated results in deviations from ideal speedup. The MO-based terms also require a significant number of cache unfriendly rearrangements of the T_2 amplitudes. These operations, similar to the four-virtual term, stress the memory bandwidth of the system and result in less than ideal scalability.

The internode scalability of the MP-CCSD method was measured as the number of nodes was increased, while the number of processors per node (PPN) was kept fixed. The internode scalability of the AO driven four-virtual term (CCSD-AO) is extremely good (Table 10), i.e., the parallel efficiency measured on one node stays approximately the same as the number of nodes is increased. This high degree of internode scalability is expected because very little communication is required relative to the amount of computational effort needed for the four-virtual term. The internode speedup is expected to extend well beyond three nodes, since the four-virtual term was modeled upon, and has a better computational vs communication ratio than the direct four-index integral transformation.

The internode scalability of the MO based terms suffers due to a low computation vs communication ratio. As mentioned earlier, the MO terms of the MP-CCSD method require a high degree of synchronization. Some of these synchronization points in the MO based MP-CCSD algorithm are collective operations which require a considerable amount of network communication. The lower computation vs communication ratio resulting from higher internode communication, combined with smaller computational workloads, significantly reduces the internode scalability of the MO based terms in the MP-CCSD program.

Despite the poor scaling of the MO-based terms, reasonable overall scalability is achieved for the MP-CCSD algorithm due to the highly scalability and overwhelmingly dominant four-virtual term. On a single processor, 88% of the execution time of the MP-CCSD algorithm was spent calculating the four-virtual term in the benchmark calculation. The outlook for the MP-CCSD algorithm for larger calculations is good, since the four-virtual term becomes increasingly dominant for larger calculations.

The performance of the triples (T) correction in the MP-CCSD(T) algorithm falls in between that of the four-virtual term and the MO-based terms of the MP-CCSD algorithms. Similar to the four-virtual term, the MP-(T) algorithm scales well as the number of nodes is increased; i.e., the efficiency does not change considerably as the number of nodes is increased for a given number of processes per node (PPN). This is expected due to the general independence of the work distribution. The intranode scaling of the (T) part suffers in the benchmark calculation from intranode synchronization and a relatively small value of N_v. That is $N_v = 408$ is at the low end of the range of N_v for which the algorithm was designed ($300 < N_v < 1000$), and, as such, the computational effort needed to evaluate the intranode parallel DGEMMs does not scale well because the subdivided DGEMMs evaluated per process are too small in size to gain a significant advantage from the highly optimized BLAS library. In the benchmark calculations, the parallel efficiency drops to just under 90% for PPN = 2, 65–70% for PPN = 4, and approximately 50% for PPN = 8 (Table 10). Larger values of N_v would provide a greater amount of computational work, and better scaling is expected.

Overall, the MP-CCSD(T) routine is dominated by two key terms: the four-virtual term and the (T) term. The intranode scaling of both of these terms is the major performance limiting variable. However, based on a fixed number of processors per node, the internode scaling is very good. Therefore, in general, a constant speedup is expected as the number of nodes is increased, even though this speedup is less than ideal due to the less than desirable intranode scaling.

Future Enhancements. The four-virtual term was designed based on the premise that quality disk I/O would not be generally available, so the method, as presented, is a fully direct algorithm. This decision was deliberate since many of the next-generation MP platforms may not have local scratch disks. However, a considerable saving in the cost of recalculating the AO integrals might be achieved by making use of a local scratch disk to store integrals or intermediates. One way in which a local scratch disk could be utilized to reduce the computational cost of recalculating the AO integrals would be to selectively store those sets of half-transformed integrals that are the *most* expensive to recalculate. Using the angular momentum quantum number (l) for the basis set shells ν and o, then for each set of ν and o in which the sum $l(\nu) + l(\mathrm{o})$ is larger than a user defined input parameter, the half-transformed integrals for the set of ν and o would be saved on the local disk during the first CCSD iteration. Subsequent CCSD iterations process all "local" sets of half-transformed integrals stored on a disk before processing the remaining sets of half-transformed integrals that must be calculated directly. There are a variety of ways in which load balancing might be achieved in such a scheme. One method would be to statically distribute the disk-based tasks while dynamically distributing the direct tasks. This would ensure that a similar amount of scratch disk is used on each node, while the dynamically distributed direct task would compensate for any potential load imbalances from the disk-based portion of the algorithm.

The limited scalability of the MO-based terms of the MP-CCSD method represents one of the major limitations in the current MP-CCSD algorithm. Each CCSD iteration performs the computationally demanding four-virtual term using every parallel process, and, when complete, every parallel process is then used to calculate the MO-based terms. Since the four-virtual term scales extremely well with the number of parallel processes, it is desirable to utilize a large number of CPUs to gain significant computational speedup. However, since the MO-based terms reach asymptotic scaling with significantly fewer processes, performing these operations sequentially results in a loss of efficiency due to the MO-based terms. To compensate for this limitation, the MO-based terms could be calculated concurrently with the AO-based terms. Using n nodes to calculate the MO-based terms, where n is the maximum number of nodes for which the MO-based terms achieve better than 50–75% parallel efficiency, the remaining nodes would then immediately begin work on the computationally dominant AO-based terms. Since the AO-based tasks are so computationally dominant, the n nodes used to calculate the MO-based terms could potentially finish before the AO-based terms are completed. In that case, those nodes would assist in the completion of the AO-based terms. This scheme would maximize the efficiency of the MP-CCSD algorithm.

Finally, improvements in the intranode scaling would benefit every step in the MP-CCSD(T) algorithm. However, in terms of an overall reduction in wall time, the biggest computational saving could be gained by improving the intranode performance of the MP-(T) algorithm. One means of improving the intranode performance of the MP-(T) algorithm (and also the intranode MP-CCSD algorithm) is to explore the use of a shared-memory model based on threads rather than processes. Thread-based models like OpenMP[38] and/or POSIX threads (Pthreads) offer a greater set of tools which are generally more robust and better performing than the limited capabilities of the System V model. Improved synchronization routines and better tuning of the intranode portion of the (T) algorithm should result in the biggest overall performance improvements.

VII. Conclusions

The MP-CCSD(T) algorithm was shown to achieve reasonable scalability for chemically interesting systems, i.e., water hexamer. The most computationally challenging portions of the algorithm, the four-virtual term and the triples corrections, achieve good internode scalability, which implies that the performance will scale well up to a large number of nodes. In general, the intranode scalability for both the MP-CCSD and MP-(T) was found to be less than optimal. However, it was only the use of the node-based model that provided the ability to perform these calculations by making it possible to store all of the various data structures. Careful consideration of the data and storage model is as crucial to the algorithm design as is CPU scaling.

The CCSD(T) calculations on isomers of water hexamer show good agreement between the CCSD(T) and MP2 methods, while the CCSD method predicts significantly worse binding energies than either CCSD(T) or MP2. While the differences between the CCSD(T) and MP2 methods are small, these differences could be important at the CBS limit or for larger water clusters, since the geometric isomers are themselves very similar in energy. Diffuse functions on the hydrogen atoms are important for calculating accurate binding energies; however, the contributions of these diffuse functions to the total energy of higher level methods, like CCSD(T), can be accurately estimated using energy differences from calculations performed at a lower level of theory, e.g., MP2.

Overall, the MP-CCSD(T) algorithm offers a node-based parallel algorithm designed to take advantage of modern cluster of SMPs. With the ever increasing trend toward more intranode compute power, most notably with the advent of multicore processors, the distinction between internode and intranode parallelism will become more important. The present work provides an initial analysis of how effectively this dual-level parallelism can be applied to modern state-of-the-art ab initio methods. While further optimizations to improve the algorithm, especially the intranode portions, should be considered, the MP-CCSD(T) method presented here is capable of calculating CCSD(T) energies for a system up to approximately 1000 basis functions in a massively parallel environment.

Acknowledgment. The authors are grateful for many illuminating discussions with Professor Alistair Rendell. The authors thank the Scalable Computing Laboratory of the Ames Laboratory for the generous allotment of computer resources allocated for use on this project. The authors also thank Professor Donald Truhlar and Ms. Erin Dalke for helpful discussion regarding basis sets, in particular the aug′-cc-pVTZ basis set. This work was supported by an ITR (Information Technology Research) grant from the National Science Foundation.

References

(1) Cízek, J.; Paldus, J. *Int. J. Quantum Chem.* **1971**, *5*, 359.

(2) Cízek, J. *Adv. Chem. Phys.* **1969**, *14*, 35.

(3) Cízek, J. *J. Chem. Phys.* **1966**, *45*, 4256.

(4) Purvis, G. D., III; Bartlett, R. J. *J. Chem. Phys.* **1982**, *76*, 1910.

(5) Raghavachari, K.; Trucks, G. W.; Pople, J. A.; Head-Gordon, M. *Chem. Phys. Lett.* **1989**, *157*, 479.

(6) Scuseria, G. E.; Lee, T. J. *J. Chem. Phys.* **1990**, *93*, 5851.

(7) Hoffmann, M. R.; Schaefer, H. F., III. *Adv. Quantum Chem.* **1986**, *18*, 207.

(8) Geertsen, J.; Rittby, M.; Bartlett, R. J. *Chem. Phys. Lett.* **1989**, *164*, 57.

(9) Stanton, J. F.; Bartlett, R. J. *J. Chem. Phys.* **1993**, *98*, 7029.

(10) Comeau, D. C.; Bartlett, R. J. *Chem. Phys. Lett.* **1993**, *207*, 414.

(11) Piecuch, P.; Bartlett, R. J. *Adv. Quantum Chem.* **1999**, *34*, 295.

(12) Kowalski, K.; Piecuch, P. *J. Chem. Phys.* **2004**, *120*, 1715.

(13) Krylov, A. I. *Chem. Phys. Lett.* **2001**, *338*, 375.

(14) Krylov, A. I.; Sherrill, C. D. *J. Chem. Phys.* **2002**, *116*, 3194.

(15) Kowalski, K.; Piecuch, P. *J. Chem. Phys.* **2000**, *113*, 18.

(16) Piecuch, P.; Wloch, M.; Gour, J. R.; Kinal, A. *Chem. Phys. Lett.* **2005**, *418*, 463.

(17) Scuseria, G. E.; Lee, T. J.; Schaefer, H. F., III. *Chem. Phys. Lett.* **1986**, *130*, 236.

(18) Scuseria, G. E.; Scheiner, A. C.; Lee, T. J.; Rice, J. E.; Schaefer, H. F., III. *J. Chem. Phys.* **1987**, *86*, 2881.

(19) Lee, T. J.; Rice, J. E. *Chem. Phys. Lett.* **1988**, *150*, 406.

(20) Scuseria, G. E.; Janssen, C. L.; Schaefer, H. F., III. *J. Chem. Phys.* **1988**, *89*, 7382.

(21) Stanton, J. F.; Gauss, J.; Watts, J. D.; Bartlett, R. J. *J. Chem. Phys.* **1991**, *94*, 4334.

(22) Schmidt, M. W.; Baldridge, K. K.; Boatz, J. A.; Elbert, S. T.; Gordon, M. S.; Jensen, J. H.; Koseki, S.; Matsunaga, N.; Nguyen, K. A.; et al. *J. Comput. Chem.* **1993**, *14*, 1347.

(23) MOLPRO is a package of ab initio programs written by H.-J. Werner, P. J. Knowles, R. Lindh, F. R. Manby, M. Schütz, P. Celani, T. Korona, G. Rauhut, R. D. Amos, A. Bernhardsson, A. Berning, D. L. Cooper, M. J. O. Deegan, A. J. Dobbyn, F. Eckert, C. Hampel, G. Hetzer, A. W. Lloyd, S. J. McNicholas, W. Meyer, M. E. Mura, A. Nicklaβ, P. Palmieri, R. Pitzer, U. Schumann, H. Stoll, A. J. Stone, R. Tarroni, and T. Thorsteinsson.

(24) ACES II is a program product of the Quantum Theory Project, University of Florida. Authors: J. F. Stanton, J. Gauss, J. D. Watts, M. Nooijen, N. Oliphant, S. A. Perera, P. G. Szalay, W. J. Lauderdale, S. A. Kucharski, S. R. Gwaltney, S. Beck, A. Balková D. E. Bernholdt, K. K. Baeck, P. Rozyczko, H. Sekino, C. Hober, and R. J. Bartlett. Integral packages included are VMOL (J. Almlöf and P. R. Taylor); VPROPS (P. Taylor); and ABACUS (T. Helgaker, H. J. Aa. Jensen, P. Jørgensen, J. Olsen, and P. R. Taylor).

(25) Kong, J.; White, C. A.; Krylov, A. I.; Sherrill, D.; Adamson, R. D.; Furlani, T. R.; Lee, M. S.; Lee, A. M.; Gwaltney, S. R.; Adams, T. R.; Ochsenfeld, C.; Gilbert, A. T. B.; Kedziora, G. S.; Rassolov, V. A.; Maurice, D. R.; Nair, N.; Shao, Y.; Besley, N. A.; Maslen, P. E.; Dombroski, J. P.; Daschel, H.; Zhang, W.; Korambath, P. P.; Baker, J.; Byrd, E. F. C.; Van Voorhis, T.; Oumi, M.; Hirata, S.; Hsu, C.-P.; Ishikawa, N.; Florian, J.; Warshel, A.; Johnson, B. G.; Gill, P. M. W.; Head-Gordon, M.; Pople, J. A. *J. Comput. Chem.* **2000**, *21*, 1532.

(26) Crawford, T. D.; Sherrill, C. D.; Valeev, E. F.; Fermann, J. T.; King, R. A.; Leininger, M. L.; Brown, S. T.; Janssen, C. L.; Seidl, E. T.; Kenny, J. P.; Allen, W. D. *PSI*, *3.2*; 2003.

(27) Aprà, E.; Windus, T. L.; Straatsma, T. P.; Bylaska, E. J.; de Jong, W. A.; Hirata, S.; Valiev, M.; Hackler, M.; Pollack, L.; Kowalski, K.; Harrison, R. J.; Dupuis, M.; Smith, D. M. A.; Nieplocha, J.; Tipparaju, V.; Krishnan, M.; Auer, A. A.; Brown, E.; Cisneros, G.; Fann, G.; Fruchtl, H.; Garza, J.; Hirao, K.; Kendall, R. A.; Nichols, J.; Tsemekhman, K.; Wolinski, K.; Anchell, J.; Bernholdt, D. E.; Borowski, P.; Clark, T.; Clerc, D.; Daschel, H.; Deegan, M.; Dyall, K. G.; Elwood, D.; Glendening, E. D.; Gutowski, M.; Hess, A.; Jaffe, J.; Johnson, B.; Ju, J.; Kobayashi, R.; Kutteh, R.; Lin, Z.; Littlefield, R. J.; Long, X.; Meng, B.; Nakajima, T.; Niu, S.; Rosing, M.; Sandrone, G.; Stave, M.; Taylor, H.; Thomas, G.; van Lenthe, J.; Wong, A.; Zhang, Z. *NWChem, A Computational Chemistry Package for Parallel Computers, Version 4.7*; Pacific Northwest National Laboratory: Richland, Washington 99352-0999, U.S.A., 2005.

(28) Kendall, R. A.; Aprà, E.; Bernholdt, D. E.; Bylaska, E. J.; Dupuis, M.; Fann, G. I.; Harrison, R. J.; Ju, J.; Nichols, J. A.; Nieplocha, J.; Straatsma, T. P.; Windus, T. L.; Wong, A. T. *Comput. Phys. Commun.* **2000**, *128*, 260.

(29) Dalton, a molecular electronic structure program, Release 2.0, 2005. See http://www.kjemi.uio.no/software/dalton/dalton.html (accessed May 2007).

(30) Frisch, M. J.; Trucks, G. W.; Schlegel, H. B.; Scuseria, G. E.; Robb, M. A.; Cheeseman, J. R.; Montgomery, J. A., Jr.; Vreven, T.; Kudin, K. N.; Burant, J. C.; Millam, J. M.; Iyengar, S. S.; Tomasi, J.; Barone, V.; Munnucci, B.; Cossi, M.; Scalmani, G.; Rega, N.; Petersson, G. A.; Nakatusjui, H.; Hada, M.; Ehara, M.; Toyota, K.; Fukuda, R.; Hasegawa, J.; Ishida, M.; Nakajima, T.; Honda, Y.; Kitao, O.; Nakai, H.; Klene, M.; Li, X.; Knox, J. E.; Hratchian, H. P.; Cross, J. B.; Bakken, V.; Adamo, C.; Jaramillo, J.; Gromperts, R.; Stratmann, R. E.; Yazyev, O.; Austin, A. J.; Cammi, R.; Pomelli, C.; Ochterski, J. W.; Ayala, P. Y.; Morokuma, K.; Voth, G. A.; Salvador, P.; Foresman, J. B.; Ortiz, J. V.; Cui, Q.; Baboul, A. G.; Clifford, S.; Cioslowski, J.; Stefanov, B. B.; Lui, G.; Liashenko, A.; Piskorz, P.; Komaromi, I.; Martin, R. L.; Fox, D. J.; Keith, T.; Al-Laham, A.; Peng, C. Y.; Nanyakkara, A.; Challacombe, M.; Gill, P. M. W.; Johnson, B.; Chen, W.; Wong, M. W.; Gonzalez, C.; Pople, J. A. *Gaussian 03, Revision C.02*; Gaussian, Inc.: Wallingford, CT, 2004.

(31) Olson, R. M.; Varganov, S.; Gordon, M. S.; Metiu, H.; Chretien, S.; Piecuch, P.; Kowalski, K.; Kucharski, S. A.; Musial, M. *J. Am. Chem. Soc.* **2005**, *127*, 1049.

(32) Harrison, R. J. *Int. J. Quantum Chem.* **1991**, *40*, 847.

(33) *SHMEM Library Reference*, Cray Research Incorporated.

(34) TheMPIForum, MPI: a message passing interface. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, ACM Press: Portland, OR, United States, 1993.

(35) Nieplocha, J.; Harrison, R. J.; Littlefield, R. J. *J. Supercomput.* **1996**, *10*, 169.

(36) Olson, R. M.; Schmidt, M. W.; Gordon, M. S.; Rendell, A. P. Enabling the Efficient Use of SMP Clusters: The GAMESS/DDI Approach. In *Supercomputing, 2003 ACM/IEEE Conference*, Phoenix, AZ, 2003; p 41.

(37) Fletcher, G. D.; Schmidt, M. W.; Bode, B. M.; Gordon, M. S. *Comput. Phys. Commun.* **2000**, *128*, 190.

(38) Dagum, L.; Menon, R. *IEEE Comput. Sci. Eng.* **1998**, *5*, 46.

(39) Rendell, A. P.; Lee, T. J.; Komornicki, A. *Chem. Phys. Lett.* **1991**, *178*, 462.

(40) Rendell, A. P.; Lee, T. J.; Lindh, R. *Chem. Phys. Lett.* **1992**, *194*, 84.

(41) Rendell, A. P.; Guest, M. F.; Kendall, R. A. *J. Comput. Chem.* **1993**, *14*, 1429.

(42) Kobayashi, R.; Rendell, A. P. *Chem. Phys. Lett.* **1997**, *265*, 1.

(43) Janowski, T.; Ford, A. R.; Pulay, P. Abstracts of Papers, 232nd ACS National Meeting, San Francisco, CA, United States, Sept. 10−14, 2006, 2006, PHYS.

(44) Koch, H.; Christiansen, O.; Kobayashi, R.; Jorgensen, P.; Helgaker, T. *Chem. Phys. Lett.* **1994**, *228*, 233.

(45) Koch, H.; Sanchez de Meras, A.; Helgaker, T.; Christiansen, O. *J. Chem. Phys.* **1996**, *104*, 4157.

(46) Rendell, A. P.; Lee, T. J. *J. Chem. Phys.* **1994**, *101*, 400.

(47) Ford, A. R.; Janowski, T.; Pulay, P. *J. Comput. Chem.* **2007**, *28*, 1215.

(48) Auer, A.; Baumgartner, G.; Bernholdt, D. E.; Bibireata, A.; Choppella, V.; Cociorva, D.; Gao, X.; Harrison, R. J.; Krishnamoorthy, S.; Krishnan, S.; Lam, C.-C.; Nooijen, M.; Pitzer, R. M.; Ramanujam, J.; Sadayappan, P.; Sibiryakov, A. *Mol. Phys.* **2006**, *104*, 211.

(49) Hirata, S. *J. Phys. Chem. A* **2003**, *107*, 9887.

(50) Piecuch, P.; Hirata, S.; Kowalski, K.; Fan, P.-D.; Windus, T. L. *Int. J. Quantum Chem.* **2005**, *106*, 79.

(51) Baumgartner, G.; Auer, A.; Bernholdt, D. E.; Bibireata, A.; Choppella, V.; Cociorva, D.; Gao, X.; Harrison, R. J.; Hirata, S.; Krishnamoorthy, S.; Krishnan, S.; Lam, C.; Lu, Q.; Nooijen, M.; Pitzer, R. M.; Ramanujam, J.; Sadayappan, P.; Sibiryakov, A. Synthesis of High-Performance Parallel Programs for a Class of Ab Initio Quantum Chemistry Models. In *Proceedings of the IEEE*; 2005.

(52) Piecuch, P.; Kucharski, S. A.; Kowalski, K.; Musial, M. *Comput. Phys. Commun.* **2002**, *149*, 71.

(53) Kendall, R. A.; Dunning, T. H., Jr.; Harrison, R. J. *J. Chem. Phys.* **1992**, *96*, 6796.

(54) Moller, C.; Plesset, M. S. *Phys. Rev.* **1934**, *46*, 618.

(55) Dunning, T. H., Jr. *J. Chem. Phys.* **1989**, *90*, 1007.

(56) Xantheas, S. S.; Burnham, C. J.; Harrison, R. J. *J. Chem. Phys.* **2002**, *116*, 1493.

(57) Feller, D. *J. Chem. Phys.* **1992**, *96*, 6104.

(58) Crawford, T. D.; Schaefer, H. F., III. *Rev. Comput. Chem.* **2000**, *14*, 33.

(59) Piecuch, P.; Kowalski, K.; Pimienta, I. S. O.; McGuire, M. J. *Int. Rev. Phys. Chem.* **2002**, *21*, 527.

(60) Urban, M.; Noga, J.; Cole, S. J.; Bartlett, R. J. *J. Chem. Phys.* **1985**, *83*, 4041.

(61) Fletcher, G. D.; Schmidt, M. W.; Gordon, M. S. *Adv. Chem. Phys.* **1999**, *110*, 267.

(62) Wong, A. T.; Harrison, R. J.; Rendell, A. P. *Theor. Chim. Acta* **1996**, *93*, 317.

(63) Olson, R. M.; Gordon, M. S. *J. Chem. Phys.*, in press.

(64) Bentz, J. L.; Olson, R. M.; Gordon, M. S.; Schmidt, M. W.; Kendall, R. A. *Comput. Phys. Commun.* **2007**, *176*, 589.

(65) Kudo, T.; Gordon, M. S. *J. Phys. Chem. A* **2001**, *105*, 11276.

(66) Day, P. N.; Pachter, R.; Gordon, M. S.; Merrill, G. N. *J. Chem. Phys.* **2000**, *112*, 2063.

(67) Dunning, T. H., Jr.; Hay, P. J. In *Methods of Electronic Structure Theory*; Plenum Press: New York, 1977; p 1.

(68) Kim, J.; Kim, K. S. *J. Chem. Phys.* **1998**, *109*, 5886.

(69) Csonka, G. I.; Ruzsinszky, A.; Perdew, J. P. *J. Phys. Chem. B* **2005**, *109*, 21471.

CT600366K