Parallel implementation of electronic structure energy, gradient, and Hessian calculations

V. Lotrich, N. Flocke, M. Ponton, A. D. Yau,[a)] A. Perera, E. Deumens,[b)] and R. J. Bartlett
Aces QC, Gainesville, Florida 32605, USA

(Received 15 February 2007; accepted 15 April 2008; published online 20 May 2008)

ACES III is a newly written program in which the computationally demanding components of the computational chemistry code ACES II [J. F. Stanton *et al.*, Int. J. Quantum Chem. **526**, 879 (1992); [ACES II program system, University of Florida, 1994] have been redesigned and implemented in parallel. The high-level algorithms include Hartree–Fock (HF) self-consistent field (SCF), second-order many-body perturbation theory [MBPT(2)] energy, gradient, and Hessian, and coupled cluster singles, doubles, and perturbative triples [CCSD(T)] energy and gradient. For SCF, MBPT(2), and CCSD(T), both restricted HF and unrestricted HF reference wave functions are available. For MBPT(2) gradients and Hessians, a restricted open-shell HF reference is also supported. The methods are programed in a special language designed for the parallelization project. The language is called super instruction assembly language (SIAL). The design uses an extreme form of object-oriented programing. All compute intensive operations, such as tensor contractions and diagonalizations, all communication operations, and all input-output operations are handled by a parallel program written in C and FORTRAN 77. This parallel program, called the super instruction processor (SIP), interprets and executes the SIAL program. By separating the algorithmic complexity (in SIAL) from the complexities of execution on computer hardware (in SIP), a software system is created that allows for very effective optimization and tuning on different hardware architectures with quite manageable effort. © *2008 American Institute of Physics*. [DOI: 10.1063/1.2920482]

I. INTRODUCTION

The methods of computational chemistry have become standard tools used on a daily basis by the practicing chemist rather than special tools with use limited to theoretical chemists; especially, the self-consistent field (SCF) method, the computationally similar density functional theory (DFT) method, are widely used due to their effectiveness of quality of results in relation to the cost of obtaining them. There are also available very efficient parallel implementations for these methods. The more traditional methods for describing correlation effects in electronic structure theory, such as second-order many-body perturbation theory [MBPT(2)] [if the Møller–Plesset partitioning is used, this is also known as the second-order Møller–Plesset (MP2)], coupled cluster theory at the singles and doubles level, CCSD, CCSD(T), and higher orders,[1,2] are computationally much more demanding. As high-level CC theory results represent the best reference results today,[3,4] it is critical to have efficient parallel implementations of these methods. We review the work on parallelization in this field in Sec. III.

ACES I was the first general purpose program to provide CC results. ACES I was developed from 1977 to 1989 from programs written by Bartlett and Purvis to perform MBPT, CC calculations, and some analytical gradient calculations. They initially used sparse matrix algorithms in which no integrals below a threshold were ever stored or processed, as that was thought to be a better large-molecule strategy. However, as vector processors like the FPS 164 emerged, the threshold based strategy was inconsistent with very fast vector processing.

ACES II (Refs. 5 and 6) was written starting in 1990, and in contrast to ACES I, was built upon fully exploiting Abelian point-group symmetry through D_{2h} using the algorithms of Stanton and co-workers.[7,8] It was directed at the CRAY machines of the time and achieved excellent vector performance. ACES II further added the general treatment of CC analytical gradients, CC density matrices, and EOM-CC excited, ionized, and electron attached states. Since then, the two developments of ACES II, the University of Florida form[9,5,6] and the Mainz–Austin–Budapest form,[10] have provided complimentary and unique treatments of a variety of properties.

In this paper, we report on the design and implementation of the methods listed in Table I for efficient execution on parallel computer systems with various architectures. Learning from the experience of those who have implemented these methods in parallel before, as will be discussed in Sec. III, we imposed some design requirements from the outset of our project. Instead of making the computational details, such as the ratio of processor speed over communication speed, subordinate to the theoretical chemistry and algorithmic considerations, as some have done, or of making the theoretical chemistry subordinate to computer science and engineering issues, as others have done, we chose to give equal weight to both concerns. To allow development to take

[a)]Present addressed: HPTi, Aberdeen, MD.
[b)]Electronic mail: deumens@qtp.ufl.edu.

TABLE I. Overview of capabilities implemented in ACES III. We do not have an efficient parallel ROHF SCF yet, but all correlated methods can use any reference, including Brueckner orbitals. The capability to drop core orbitals is also implemented.

Method	Energy	Gradient	Hessian
SCF	RHF, UHF	RHF, UHF, ROHF	RHF, UHF, ROHF
MBPT(2)	RHF, UHF, ROHF	RHF, UHF, ROHF	RHF, UHF, ROHF
LinCCSD	RHF, UHF, ROHF	RHF, UHF, ROHF	
CCSD	RHF, UHF, ROHF	RHF, UHF, ROHF	
CCSD(T)	RHF, UHF, ROHF	RHF, UHF, ROHF	

place in an organized fashion within such constraints, we formulated a precise separation of the two aspects and defined an interface and protocol for the two groups to interact.

The basic idea of the division of expertise is that, for floating point intensive applications like the ones under consideration, one should think in terms of blocks of numbers instead of individual numbers. Modern computer hardware has a hierarchy of speeds versus size, ranging from very high speeds for small amounts of data inside processors down to many orders of magnitude slower access to huge amounts of data on disk subsystems. Individual numbers are very inefficiently processed across this hierarchy. However, for each parallel computer architecture, a block size that can be processed much more efficiently can be found. We call these blocks *super numbers* and the basic operations on them *super instructions*.

In Sec. II, we describe some of the details of the mathematical and algorithmic aspects of the computational chemistry methods in the example of CCSD energies, as it exhibits all of the algorithmic complexity of gradient and Hessian computations as well. In particular, the data types and patterns are analogous between the algorithms developed for all methods beyond CCSD. Then, in Sec. III, we present a brief review of parallel implementations of various CC methods that have been published. We summarize the innovative algorithms and the computing technology employed and from that the motivation for our approach is derived. In Sec. IV, we describe the details of the parallel implementation of the computational chemistry methods in ACES III and of the super instruction architecture (SIA). In Sec. V, we show some performance results obtained with ACES III.

II. THEORETICAL METHODS

Although we have implemented a variety of commonly used open-and closed-shell *ab initio* methods into ACES III (see Table I), the complexity involved in the parallel implementation of these methods can be satisfactorily illustrated by considering the complete calculation of the CCSD energy. We will therefore give a brief summary of the CCSD method with emphasis on the data required to perform the calculation.

So far, we have only considered CCSD methods based on a Hartree–Fock [restricted HF (RHF) and unrestricted HF (UHF)] reference wave function $|\Phi_0\rangle = |\Phi_{\mathrm{HF}}\rangle$ which simplifies the gradient and Hessian computation but does not affect the present discussion. The coupled cluster method expresses the exact wave function of a molecule as $|\Psi\rangle = e^T|\Phi_0\rangle$, where the operator T creates all N-tuple excitations of the reference function $|\Phi_0\rangle$, N being the number of electrons of the molecule. The many-body Schrödinger equation used to compute the stationary state energies and wave functions can then be written very simply as

$$H|\Psi\rangle = He^T|\Phi_0\rangle = Ee^T|\Phi_0\rangle. \tag{1}$$

Acting from the left with the operator e^{-T} and projecting against the reference function $\langle\Phi_0|$ leads to the expression for the energy

$$\langle\Phi_0|e^{-T}He^T|\Phi_0\rangle = E, \tag{2}$$

whereas projection against an excited determinant $\langle\Psi_{ij\cdots}^{ab\cdots}|$ leads to the equations which determine the CC amplitudes.

$$\langle\Psi_{ij\cdots}^{ab\cdots}|e^{-T}He^T|\Phi_0\rangle = 0. \tag{3}$$

In Eqs. (2) and (3), expansion of the exponential leads to the terminating expansion,

$$e^{-T}He^T = H + [H,T] + \frac{1}{2}[[H,T],T] + \frac{1}{3!}[[[H,T],T],T] + \frac{1}{4!}[[[[H,T],T],T],]T], \tag{4}$$

where we use the commutator $[H,T]=HT-TH$. Since the Hamiltonian H is a two-particle quantity, the amplitude equations [Eq. (3)] are at most quartic in any T_n. Note that up to this point, we have made no restriction on T, restricting the cluster operator T to singles and doubles excitations leads to the CCSD equations, which are quadratic in the two-particle cluster operator T_2, cubic in $T_2T_1T_1$, and quartic in the one-particle cluster operator T_1. These equations (the CCSD energy and amplitude equations) are well known[4,7,8] and will not be repeated here.

In order to solve Eqs. (2) and (3), once the zeroth-order wave function is determined, the two-electron integrals must be transformed from the atomic orbital (AO) to the molecular orbital (MO) basis using the transformation coefficients determined in the SCF. The two-electron transformation can be written as

$$V_{rs}^{pq} = C_\mu^p C_\nu^q C_r^\lambda C_s^\sigma V_{\lambda\sigma}^{\mu\nu}, \tag{5}$$

where p, q, r, and s are MO labels and μ, ν, λ, and σ are AO labels, and repeated indices are summed over.

Determination of the CCSD energy and gradient consists of five steps: (1) Solve the Hartree–Fock equations. (2) Perform the two-electron integral transformation. (3) Solve the

CCSD equations to determine T amplitudes, from which the energy can be constructed. (4) Solve the complementary Λ equations to provide analytical gradients and density matrices.[11] (5) Construct the gradient using T, Λ, and the one-electron and two-electron Hamiltonian matrix elements.[12]

We now will discuss the data requirements and computational cost associated with each of these steps.

A. Hartree–Fock method

The Hartree–Fock method approximates the wave function as an antisymmetrized product of single particle functions which can be determined from a variational one-particle equation. Only one-particle (two-dimensional array) quantities and two-electron integrals (four-dimensional arrays) need to be evaluated. If the two-electron integrals are recomputed each time they are needed, then no complete two-particle quantities need to be stored, but only a portion. For a computation involving 500 to 1000 basis functions, each one-particle array requires 2–32 Mbytes, respectively. On modern computers, this amount of data easily fits into memory so that disk usage can be completely avoided in the SCF computation for all but very large molecules on very small computer systems.

B. Two-electron integral transformation

In the CCSD computation, all of the two-electron integrals are used but not necessarily in transformed form. The computation is usually done in four steps with one index being transformed at a time so that the computational cost is on the order of N^5, where N is the number of basis functions. More importantly, even if the two-electron integrals are not stored but recomputed as needed, the data that need to be stored is on the order of N^4. For example, if the number of basis functions is 500, then the transformed integrals require about 50 Gbytes of disk or memory and 1000 basis functions requires 900 Gbytes. This is a large amount of data to be stored on most computers available today and thus usually requires that the integrals be stored on disk, unlike the SCF where the entire computation can be done in memory.

C. Coupled cluster method

The equations that define the CCSD amplitudes[8] can be derived from Eq. (3) but we will not reproduce them here. Of significance to this discussion are the extra storage requirements above and beyond the two-electron integrals and the computational cost. The singles and doubles amplitude storage is small compared to the storage of the transformed integrals. For the perturbative triples,[13,14] the six-index t_{ijk}^{abc} amplitudes are not stored. For 500 and 1000 basis functions and a ratio of 5/1 of virtual to occupied orbitals, the doubles amplitudes require 12.8 and 204.8 Gbytes of storage, respectively, which is a manageable amount. The triples, if stored, would require 50 000 and 200 000 times this.

The CCSD implementation includes the direct inversion in the iterative subspace (DIIS) method[15] for accelerating the convergence of the amplitudes. Such convergence acceleration requires storage of a number of amplitude histories, increasing the data storage requirement.

D. Coupled cluster gradients

The CC functional[4] can be written as

$$E = \langle 0|(1+\Lambda)e^{-T}He^{T}|0\rangle, \tag{6}$$

with Λ operator similar to the T operator

$$\Lambda = \sum_{ia} \lambda_a^i i^\dagger a + \sum_{ijab} \lambda_{ab}^{ij} i^\dagger j^\dagger ab + \cdots . \tag{7}$$

In these equations, we use the convention that $i,j,k,\ldots,$ denote indices that range over occupied orbitals and $a,b,c,\ldots,$ range over virtual orbitals. Imposing stationarity with respect to λ_n provides the equations for T, while imposing stationarity with respect to T delivers Λ. The Λ equation is linear, unlike the equation for T and its origin lies in the interchange theorem,[3,16] and without it, analytical gradients could not be constructed for CC methods. Λ also permits the definition of CC density matrices

$$\gamma(\mathrm{PDQ}) = \langle 0|\Lambda e^{-T}p^\dagger q e^{T}|0\rangle, \tag{8}$$

$$\Gamma(pq,rs) = \langle 0|\Lambda e^{-T}p^\dagger q^\dagger rs e^{T}|0\rangle \tag{9}$$

that are critical for properties. Unlike conventional density matrices, these also exist for methods like CCSD(T) that have no wave function form.

The other elements required for CC gradients are the integral derivatives and the derivatives of the MO coefficients $c_{\mu p}$. The latter are given by

$$\frac{\partial c_{\mu p}}{\partial \chi} = \sum_q U_{qp}^{\chi} c_{\mu q}. \tag{10}$$

The MBPT or CC gradient with respect to a nuclear coordinate χ can be written as

$$\frac{\partial E}{\partial \chi} = \sum_{ab} D_{ab} f_{ab}^{(\chi)} + \sum_{ij} D_{ij} f_{ij}^{(\chi)} + \sum_{pqrs} \Gamma(pq,rs)\langle pq||rs\rangle^{\chi} - 2\sum_{pq} I'_{pq} U_{pq}^{\chi}, \tag{11}$$

where the definition of all quantities can be found in Ref. 17. The superscript χ indicates that an integral derivative is to be taken and transformed to the MO basis whereas the superscript (χ) indicates that the derivative of the Hartree–Fock density is to be omitted. In the case of the CC gradient the two-particle density $\Gamma(pq,rs)$ depends on the CC amplitudes T and on a similar set called Λ coefficients. The Λ coefficients are obtained by solving a linear set of equations similar to the CC equations for T amplitudes. The CPHF coefficients U_{pq}^{χ} only appear in the last term in Eq. (11). This is a valid equation but it can be simplified if the invariance with respect to orbital rotations within the occupied and virtual spaces is used. This allows the occupied-occupied and virtual-virtual blocks of the CPHF coefficients to be written as $-\frac{1}{2}S_{ij}^{\chi}$ and $-\frac{1}{2}S_{ab}^{\chi}$, respectively, where S_{ij}^{χ} is the derivative of the overlap integral transformed to the MO basis. Eq. (11)

still contains the virtual-occupied block of the CPHF coefficients but this block can be removed using the interchange theorem again.[3,16] This procedure was later called the *Z*-vector method[17] for CPHF. The final equation for the gradient

$$\frac{\partial E}{\partial \chi} = \sum_{pq} D_{pq} f_{pq}^{(\chi)} + \sum_{pqrs} \Gamma(pq,rs)\langle pq\|rs\rangle^{\chi} + \sum_{pq} I_{pq} S_{pq}^{\chi} \quad (12)$$

does not contain the CPHF coefficients but does contain the perturbation-independent occupied-virtual block of the relaxed density D_{ai} which must be constructed. The one particle intermediates I_{pq} and components of the reduced density matrix D_{ab}, D_{ij}, and $\Gamma(pq,rs)$ are defined in Ref. 21

Alternatively, if we consider differentiating the MBPT or CC, energy we get

$$\frac{\partial E}{\partial \chi} = \sum_{ab} D_{ab} \frac{\partial f_{ab}}{\partial \chi} + \sum_{ij} D_{ij} \frac{f_{ij}}{\partial \chi} + \sum_{pqrs} \Gamma(pq,rs) \frac{\partial \langle pq\|rs\rangle}{\partial \chi}. \quad (13)$$

Note that all derivatives appearing in Eq. (13) are complete derivatives and therefore contain the CPHF coefficients U_{pq}^{χ}. This expression requires that the CPHF coefficients be determined for each perturbation χ and is not actually implemented. It, however, is a proper starting point to determine the Hessian.

The MBPT(2) gradient is defined by Eq. (12) if I and $\Gamma(pq,rs)$ are restricted to second order.[12] The second-order $\Gamma(pq,rs)$ reduces to $\Gamma(ab,ij)$. The algorithm we have used to compute the MBPT(2) gradient calculates all contributions that depend on the integrals with three virtual indices directly. Two-electron integrals containing four virtual indices are not needed. The partial two-electron transformation to obtain all integrals with less than three virtual indices is done as described in Ref. 12. We have included the transformation within the program segment that computes the gradient.

The linear-CCSD gradient is very similar to the CCSD gradient with the following exceptions. First, LinCCSD is a Hermitian theory so the LinCCSD and Λ amplitudes are the same.[12] Therefore, no computation of Λ is required. Since the CCSD equations are linear, we have chosen to compute in a direct way all terms which depend on two-electron integrals with three and four virtual indices. Since computation of the two-particle density requires the transformed integrals with three virtual indices, we perform a transformation *after* the CCSD step to obtain them.

The equations for the perturbative triples energy and gradient can be found in Refs. 2 and 3

Our implementation of the CCSD gradient computation starts with the the SCF step immediately followed by the transformation of two-electron integrals involving three virtual indices or less. This step is followed by the two steps computating the T and Λ amplitudes, respectively. In the computation of T and Λ, we have eliminated, using the standard textbook technique, the need to perform the two-electron integral transformation involving four virtual indices by performing all computations involving these integrals directly.

As soon as a block of the density is computed, it is used to compute the corresponding parts of the one-particle intermediates. These intermediates are then contracted with the one-electron derivative integrals to obtain the contribution to the gradient. The reduced density is computed a second time for the two-electron derivative integral part. If a density contribution contains two or fewer virtual indices it is partially back-transformed to be contracted later with the derivative integrals. Gradient contributions which contain three virtual indices are backtransformed to the AO basis immediately after they are formed and contracted with the derivative integrals (computed as needed in the AO basis) to form their contribution to the gradient. The two-particle density array containing four virtual indices is computed directly and does not need to be stored.

E. MBPT(2) Hessian

We have considered two different approaches to the UHF Hessian and thus have written two different programs which yield identical results. One of these programs derives the Hessian as the derivative of the MBPT(2) gradient [Eq. (12)]. The other approach directly differentiates the energy expression twice and then removes the second-order coupled Hartree–Fock coefficients using the interchange theorem.[19] This is equivalent to differentiating Eq. (13) and then removing the second-order coupled Hartree–Fock coefficients.

Consider the first approach. Differentiating Eq. (12) leads to one expression for the Hessian that we have implemented. It depends on the derivative of D_{ai} and is only valid for the set of CPHF equations defined in Eq. (10) with the occupied-occupied and virtual-virtual blocks of CPHF coefficients given by $-\frac{1}{2}S_{ij}^{\chi}$ and $-\frac{1}{2}S_{ab}^{\chi}$. This has the consequence that the derivatives of the Fock matrix are no longer diagonal.

When we consider differentiating Eq. (13), we get an expression that is slightly more complicated but does not depend on any choice of basis (and therefore on a choice of CPHF coefficients). Using the second-order CPHF coefficients it is convenient at this point to replace the occupied-occupied and virtual-virtual blocks of the CPHF coefficients with $-\frac{1}{2}S_{ij}^{\chi}$ and $-\frac{1}{2}S_{ab}^{\chi}$, respectively. The final equation for the MBPT(2) Hessian thus obtained can be found in Ref. 18. This choice of basis also results in the derivatives of the Fock matrix having off-diagonal components.

III. REVIEW OF PARALLEL IMPLEMENTATIONS

Electronic structure methods have been implemented in efficient computer programs for a long time.[19,20] In the past two decades, many implementations have exploited parallelism available in modern computer architectures. The literature is extensive and it is beyond the scope of this section to provide a review that does justice to all contributions. However, we do want to provide an overview with sufficient detail to properly position the parallel implementation presented in this paper. The architecture of our implementation is strongly influenced by the ideas published in the literature and by the successes obtained and limitations discovered from their implementation.

We can coarsely divide high-performance strategies for modern computer codes into two approaches: algorithms and technology. This division is not perfect as there are strong overlaps, but nevertheless it will help organize the discussion of the literature.

Implementing any electronic structure theory requires a significant effort. Because the computations tax available computer hardware to the fullest, it is important to make the implementation efficient. However, the wide range in available hardware architectures has the result that an implementation that is efficient on one architecture may very well be very ineffective on another. Thus, any implementor is confronted with the challenge of finding implementations, algorithms and technology, that are both portable and adaptable to many architectures.

Some groups, lead by the NWChem development team,[21] have developed a sophisticated technology to distribute and manipulate data on a distributed memory computer architecture. This technology is implemented in a library component like global arrays (GA),[22,23] which is used in NWCHEM,[24] and MOLCAS[25] and COLUMBUS.[26] Similar technology is embodied in the distributed data interface[27] (DDI) of GAMESS.[29] The design goals of the NWChem team were to create software that could execute efficiently on massively parallel systems with a fast interconnect for communication between processors. This led to the decision to store large data structures in distributed memory. The GA toolkit then provides a uniform programing interface for the chemist developers to use the distributed memory of many different architectures of distributed memory computer systems. The developers of GAMESS (Ref. 28) followed a similar reasoning, but chose to build a new, simpler interface called DDI (Ref. 27) rather then use the existing feature-rich GA toolkit.[22,23] However, since the original design, the DDI has grown in complexity as well, such as with the recent addition of support for shared memory parallelism.[29] The NWChem developers have added a disk resident array interface[30] to their toolbox. The developers of MPQC (Ref. 31) have implemented parallelism for the computation of MP2-R12 (Ref. 32) and local MP2 (Ref. 33) energies.

One side effect of the pure distributed memory design is that problems with larger molecules can only run on large numbers of processors with fast interconnects. Such computer systems are available, but many research groups have more ready access to small clusters with 16 to 128 CPUs with slower interconnects such as 100BT or Gigabit Ethernet. Several groups, for example Pulay and co-workers[34–37] and Szalewicz and co-workers[38] and the developers of TURBOMOLE,[39] have opted to implement their algorithms using the basic PVM or MPI technology, foregoing tools such as the GA or DDI. They designed their code from the start to run effectively on smaller clusters of nodes with local disks and slower inter-node-communication performance. This leads to the consideration of different algorithms. It allows them to run significantly larger problems on small clusters than on a single CPU by using disk based algorithms with a (small) number of disks local to the nodes in the cluster. Distributed memory codes like NWCHEM or GAMESS would typically require much larger numbers of CPUs and a better interconnect to run these same problems.

After this brief review of the technology used by the developer teams, we now analyze the different efforts from the point of view of the algorithms used.

The first observation that is made by every team is that the computation of electron-repulsion integrals is a naturally parallel task. Every computer program listed above computes them in parallel. For some electronic structure methods, such as the Hartree–Fock self-consistent field (SCF) method, a so-called direct method is used: the integrals are computed and immediately used. For other methods such as MBPT(2) the integrals are computed during the integral transformation step, as explained for example by Saebo and Pulay.[40]

The second observation made by most is that many computational steps in electronic structure methods have a matrix multiplication as a major component. Matrix multiplications can be executed very efficiently especially on shared memory machines if the matrices can be broken into independent pieces so that no corruption of data can result from multiple CPUs computing on the same data at the same time. Examples are the steps during the integral transformation from AOs to MOs as used in several programs[38,41] and the perturbative triples part in the CCSD(T) method.[29,24,20] An increasing number of authors[37,33] use an organization of data in blocks, i.e., tiled indices, that are then more readily processed optimally with these matrix operations.

Some authors[30,39] make explicit their design decision to target problems with a certain number of atoms and basis functions, and from that derive which data structures can be replicated on each node and which must be distributed. Some[38,39,36,42] build in the capability to store certain data structures on disk, as was customary for serial implementations. On clusters of PCs, where each node has a local disk, this works quite well, but on high-end systems, like the SP4, with a parallel file system, like GPFS, the resulting I/O can cause a performance bottleneck.[38] Bentz *et al.* consider a refinement of the data distribution by distinguishing between CPUs on a single node and CPUs on different nodes. This effort is worthwhile in view of the ubiquitous presence of clusters with multi-CPU nodes. Recognizing the “closeness” of CPUs, GAMESS can tackle larger problems as CPUs can access data owned by other CPUs that are on the same node at the fast speeds of local memory, thus requiring less replication of data.

Improvements in algorithms generally give better performance increases than technology changes. For example the algorithms introduced by Pulay and co-workers reduce the number of integrals to be computed without loss of accuracy by sophisticated screening techniques.[34,36,40] Another example is the resolution-of-the-identity method for MP2 as implemented, for example, in TURBOMOLE.[39] For the computational chemist, the total time to solution and the total resources, CPUs, memory, disk, are what matters. However, for developers it is important to know which of the two contributed most to the observed efficiency: algorithms or technology. More often than not a clever change in algorithm contributes more, often an order of magnitude, than any technology. For example, the use of localized orbitals and efficient integral screening[34,36] or of the resolution-of-the-identity method[39] has a much larger impact on the feasibility and execution time of MP2 computations than the use of array files,[35] disk resident arrays,[36] global arrays,[22,23] or DDI[27] The proper comparison to exhibit the value of parallel technology requires comparing calculations that manage the same amount of data and perform the same number of floating point operations. After reduction in the amount of data that must be moved and the number of operations that must be performed by the choice of a novel algorithm, the role of parallel technology is less than might be expected. Straightforward use of MPI will produce code that performs well.[39,36] Hättig *et al.* give a nice analysis, similar to what many authors have done before them, of the balance between the time needed to communicate integrals between nodes and the time it takes to recompute them on every node. This balance is different on departmental clusters with Gigabit Ethernet and supercomputers with fast interconnects. They implemented code for both strategies.

The approach discussed in this paper is not an alternative to these methods but rather a natural evolution of them. We try to design an architecture for the software that can be adapted to a wide range of computer architectures without requiring large human effort and that allows any of the algorithms published in the literature to be implemented quickly.

IV. PARALLEL ARCHITECTURE OF ACES III

We now explain the detailed architecture of the new, parallel components of ACES III. ACES III is the latest implementation of the ACES program system and includes the latest version of the serial ACES II as a whole. The new components of ACES III are replacements for some parts of ACES II, namely, some of the compute intensive parts that are able to run on parallel computers with shared or distributed memory with quite respectable scaling, as will be shown in Sec. V. Thus, ACES III contains the familiar serial executables xjoda, xvscf, xvcc, etc in addition to the new parallel executable xaces3. For example, to perform a geometry optimization with ACES III in parallel, one should run the executable xaces3.

It is possible to describe ACES III as a runtime environment to execute, in an efficient way using all levels of parallelism of the given hardware platform, programs, written in a high-level programming language that implement the algorithms of electronic structure methods.

We call the high-level programing language SIAL for super instruction assembly language, and the runtime environment SIP for super instruction processor. Both of these will be explained in Sec. IV B in detail. We refer to a design using this division of software into SIAL and SIP as the super instruction architecture (SIA).

One of the main goals of the design of ACES III was to clearly separate two domains of expertise, both exhibiting considerable complexity. The first is the domain of theoretical chemistry and applied mathematics of solving the equations at the core of the methods of electronic structure theory: SCF, MBPT(2), CCSD, etc. Application of these methods requires specification of various algorithms to solve numerical problems. The second is the domain of computer science and engineering of implementing the algorithms for efficient execution on modern computer hardware. The SIAL computer language allows the "quantum chemist programmers" to accurately and concisely express the algorithm without controlling the details of the computation and communication. That is left to the "computer engineering programmers" in charge of the SIP.

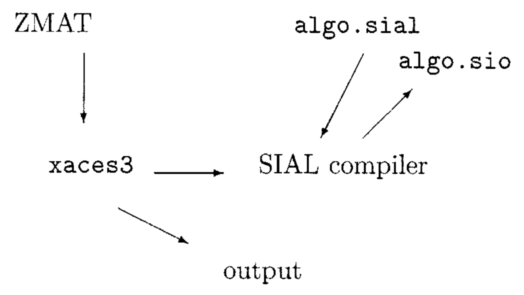

FIG. 1. Connection between the files involved in running the parallel executable xaces3 in ACES III. See text for details.

One result of this precise division of expertise is that a lot of flexibility is left to optimize and tune the performance of the execution (in SIP) without the need to alter the algorithm (in SIAL). We also found that some algorithms perform better than others depending on the molecule and the computer system. Therefore, it is to the advantage of the end-user of ACES III that both algorithms are available and that their use is easily controlled. In the current implementation the control requires manual intervention by the user to change the input file, an automatic selection of the optimal algorithm is planned.

A. Components of ACES III

All parallel capabilities are programed, in accordance with the super instruction design to be explained in the next section, in the super instruction assembly language or SIAL. The parallel executable xaces3 reads, interprets, and executes the SIAL programs. For this reason, this executable is called the super instruction processor or SIP. A high-level overview of the connection between the files involved in running xaces3 is shown in Fig. 1.

The SIP xaces3 reads the ACES III input file. It is identical to the ACES II input file with the addition of one new section with header *SIP. This section contains directives for the parallel execution. It also contains the name of the SIAL programs to execute. The SIAL programs can be specified in source form, for example, algo.sial, or in object form. The SIAL compiler is called as a subroutine from xaces3. It takes a program expressing the algorithm in the SIAL language contained in a file algo.sial and produces the compiled object form of the program in the file algo.sio. This object code is then loaded into the executable xaces3 for execution. There is also an executable called sial to compile SIAL programs.

Next we consider the architecture of the SIP xaces3 itself. xaces3 is a parallel MPI program that consists of multiple tasks, with each task being a single-threaded or, on some computer architectures, a POSIX multithreaded process. The SIP has the following software components:

```
PARDO M,N,L,S
  COMPUTE_INTEGRALS V(M,N,L,S)
  DO I
    DO J
      REQUEST T(L,S,I,J)
      TEMP(M,N,I,J) = V(M,N,L,S) * T(L,S,I,J)
      PREPARE R(M,N,I,J) += TEMP(M,N,I,J)
    ENDDO J
  ENDDO I
ENDPARDO M,N,L,S
```

FIG. 2. SIAL source code for a basic contraction of coupled cluster amplitudes T with two-electron integrals V. See text for explanation of the language.

(1) Master component is a component coordinating the work to be done by all tasks. This component executes in the master task during initialization. It reads the ACES III input file; loads the SIAL object code, possibly after compiling the SIAL source code; analyzes the work to be done and sets up tables defining the work to be done by each task in the parallel program.
(2) Worker component is a component for executing basic chunks of work in the form of super instructions. This component needs to have asynchronous, one-sided access to distributed data.
(3) Communication component is a component for communication of basic data elements, blocks, between the cooperating tasks.
(4) I/O server component is a component for storing and retrieving large amounts of data to a large disk storage system as opposed to distributed memory.

B. Super instruction architecture

To explain the super instruction way of designing parallel software, consider the example of one important term in the CCSD electronic structure method. We show the term, usually referred to as the AO ladder term, with the CC amplitudes partially in the AO basis and partially in the MO basis because this allows the use of the integrals in the AO basis. This formulation is most suitable for a direct implementation of this term,

$$R_{ij}^{\mu\nu} = \sum_{\lambda\sigma} V_{\lambda\sigma}^{\mu\nu} t_{ij}^{\lambda\sigma}. \tag{14}$$

The SIAL program for this sum is shown in Fig. 2. We will discuss the little program in detail and introduce the important concepts of the design at the same time.

First we need to decide how to process such expressions. To get an idea of the amount of data to be processed, consider a molecule with 200 occupied orbitals and 1,000 basis functions. Then, the two-electron integrals take up 900 Gbytes and there are $200^2 \times 800^2 = 2.56 \times 10^{10}$ CCSD amplitudes $t_{ij}^{\lambda\sigma}$, representing 200 Gbytes of data. On modern computers, processing floating point numbers one at a time is very inefficient because it the processor is much faster than the memory from which the numbers must be read and to which the rsults must be returned. It has been known for may years, since the analysis done by Jack Dongarra, that it is much more efficient to process numbers in blocks of floating point numbers. A good data block is made by dividing all index ranges into segments of length 10 to 50. An index running over the segments is then a super index. Then blocks of T and V each contain 10 000 to 6 250 000 floating point numbers, or 80 Kbytes to 50 Mbytes. These are then the super numbers of the problem. The term in the CCSD equation then can be blocked as follows

$$R(M,N,I,J)_{ij}^{\mu\nu} = \sum_{LS} \sum_{\lambda\in L} \sum_{\sigma\in S} V(M,N,L,S)_{\lambda\sigma}^{\mu\nu} t(L,S,I,J)_{ij}^{\lambda\sigma}, \tag{15}$$

where the capital letters M, N, L, S, I, J indicate block indices and the lowercase letters μ, ν, λ, σ, i, j indicate indices inside each block ranging from 1 to 10 or 50. For example, $V(A,B,C,D)$ is itself a four-index matrix of size $10\times 10 \times 10 \times 10$ to $50\times 50\times 50\times 50$.

This expression then is coded in super instruction assembly language (SIAL) as the program shown in Fig. 2. This short program fragment shows the simplicity and power of the super instruction approach. The example does not use the eightfold symmetry of the two-electron integrals or the symmetry of the CC amplitudes.

The PARDO parallel-do construct indicates that many processors can execute the block of code simultaneously. Load balancing is done as in OpenMP, and several options have been implemented such as divide and conquer, round robin, etc. The loops over I and J are serial loops over super indices, i.e., segments of orbitals.

The super instruction way of thinking also calls for computing integrals in blocks. Many integral evaluation algorithms, including the Rys polynomial method newly written for ACES III,[43] compute integrals in groups of AO orbitals that are part of an angular momentum shell. In general, a block of integrals will include a few shells for each of the four AO indices. We therefore implement computation of one block of integrals as a single super instruction COMPUTE_INTEGRALS. The results is stored in a single block V(M,N,L,S). The array is declared as a temporary array, as will be explained further below.

The REQUEST fetches a block of the array T. The array is declared as a served array, see below. The program sends an asynchronous request to the server task that is responsible for the block. Each worker task has the index for all arrays and therefore knows where all data resides. The instruction does not make the processor wait. To avoid waiting times during communication, the super instruction design uses one-sided communication like SHMEM or POSIX threads combined with MPI. We have implemented and tested both. On each platform we can select the fastest implementation to execute the same SIAL programs. The super instruction processor continues to work on the next super instruction, while the communication is in progress.

The server task may find that the block is in its block cache in memory and then it will send the block immediately. If the block is not in cache, it will initiate a read request to the disk storage to fetch the block and send it as soon as the block has arrived.

While the integrals are being computed, the requested block of amplitudes T is being sent. The super instruction view of designing parallel software suggests thinking of all distributed memory of the parallel computer as accessible by read and write super instructions and to take into consideration the communication delay as part of the time needed to complete the instruction. With a communication interconnect capable of transmitting at 100 Mbytes/s, it takes less than 1 ms–0.5 s to transmit one block of T or V. The latency of the interconnect, from 100 μs for gigabit Ethernet to 1 μs for Infiniband, should be small compared to the time it takes for the processing of the data block. We will see in the next paragraph that this is often the case.

The star $*$ indicates a super instruction that contracts two blocks, one block of V with one block of T, and produces one block of R. This instruction can be implemented as a DGEMM and involves 2×100^3 to 2×2500^3 floating point operations. On a 1 GHz processor, this operation takes from 2 ms to 30 s. As part of the super instruction, the processor checks whether all blocks are available. Since the integral computation of one block is itself a significant computation, the communication of the remote block of T is likely to be complete before the integral computation ends and the $*$ instruction can proceed without delay.

The inner loop executes the PREPARE+=super instruction to accumulate the contribution into the served array R, taking care of data locking to prevent data corruption. This completes the explanation of the example program in Fig. 2.

SIAL programs achieve good performance, even as they involve a significant amount of complex messaging, by hiding communication behind computation. This is accomplished without explicit coding by the programmer. The key consideration is that each operation must take sufficiently long.

Note that the SIAL programmer has no access to the indices μ, ν, λ, σ, i, j counting inside the blocks in Eq. (15). These are only visible inside the super instructions. The time for each super instruction to complete can be tuned by changing the segment size of the indices and thus the block size. It then becomes reasonable to expect that every read from and write to some other task using MPI communication can be tuned to be less than the time for a block operation. A large portion of all communication is automatically hidden behind computation, resulting in favorable parallel performance of the program. In the example above of segment sizes of 10 to 50, the time to execute a computation super instruction and the time to execute a memory super instruction, i.e., data communication super instruction, range from similar to an order of magnitude different. It is likely that the segment size can be reduced in the larger example without jeopardizing the hiding of communication behind computation.

To perform computations at the CCSD level, SIAL supports arrays with four indices, where each index is "segmented." This means that the arrays, such as CCSD amplitudes, can be blocked or tiled with each block holding all elements with the four indices each inside a respective segment. This allows optimal use of modern computer architectures with a deep hierarchy of memory. To support higher order coupled cluster algorithms, there is a need for arrays with more indices. These additional indices are standard single-value indices, as in C or Fortran. Thus, to support triples, SIAL programs use two types of indices: segmented and single-value indices.

We close this section with a brief overview of the arrays defined in SIAL, offering the programmer a range of option for data from very local, and thus fast but limited in size, to remote and slower but larger in size. The data types are, from most local to most remote, as follows.

(1) Any array that is small enough to fit easily into the memory of a single processor is designated a static array. In a parallel computation it will be replicated across all of the processors so that it is always available. The transformation coefficients C_μ^p are an example of a static array.

(2) Some arrays such as the two-electron integrals (not transformed) are too large to be stored in memory. However, if only the portion actually being used is considered, the partial array can fit if the segment of data being used is small enough. Such arrays, which are never stored in their entirety, are designated temporary arrays. They are discarded immediately after they are used.

(3) A special case of a temporary array is a local array in which one or more of the indices is formed completely. It is used just like a temporary array but adds flexibility to the algorithm implementation.

(4) There are arrays which typically will not fit into the memory of one processor but will fit into the combined memory of a reasonable number of processors. An example of such an array is the transformed two-electron integral containing four occupied indices V_{kl}^{ij}. For the 500 and 1000 function systems described above, V_{kl}^{ij} requires 0.8 Gbyte and 12.8 Gbytes, respectively. This amount of data can be distributed over a multiple processors. This gives rise to the name distributed array. This type of array is the SIAL equivalent of the arrays in the GA toolkit.[22,23]

(5) Finally, arrays such as V_{cd}^{ab} can be very large and the programmer may want them to exist on disk to be served to a processor when requested. Because served arrays involve input-output to external storage in addition to communication, the latency of the request can be high. The current implementation uses a caching algorithm on the servers to keep blocks that are used most often in memory, thus allowing it to send the block more quickly upon request. This type of array provides the functionality of the disk resident arrays[30] and the Array Files.[35]

In a future implementation of the SIP, it will be possible to share some array types among processors in multicore nodes. Note that this will not require a change to the language or the SIAL programs.

TABLE II. Time to perform one MBPT(2) gradient calculation on the Ar_6 molecule. The basis is aug-cc-pVTZ, which results in 300 basis functions (bf) and the number of correlated occupied orbitals is 54. A UHF reference and C_1 symmetry was used. The ratio of workers/servers was 3/1 for all computations.

N_p	t	t^{ideal}	t^{ideal}/t
32	67	67	1.00
64	32	34	1.06
128	18	17	0.94
256	16	8	0.5

C. Support of multiple computer hardware architectures

We have ported the SIP xaces3 to many parallel computer architectures. We started with the IBM SP3 with SP Switch and IBM SP4 with SP Switch2. This system, because of IBM's involvement in the ASCI White program, has a working thread safe MPI implementation. We used MPI and POSIX threads.

Next we ported to a cluster of Compaq-HP SC45 nodes with a Quadrix switch. This MPI implementation does not operate error-free in a true multitheaded environment. For this reason we switched to SHMEM. This worked and performance was good. The SHMEM version was ported to the Cray SV1 and then the Cray X1. Because of process and task scheduling on the Cray X1, we had to redesign the SIP. After the redesign, the performance was acceptable, but not outstanding.

We also ported to SGI Origin 3000 system, but the system was decommissioned before we could complete the port. The port to the SGI Altix required another redesign because the MPI implementation is not thread safe and the SHMEM was slow. The redesign uses pure MPI. This actually works in the SP because the SIP is executing SIAL code in the form of individual super instructions. Between every instruction, the SIP can check whether any MPI requests have arrived and process them. Thus, a separate thread or one-sided access is not really needed. The pure MPI version has been ported to all other platforms and gives the best performance of all, even on shared memory parallel computers.

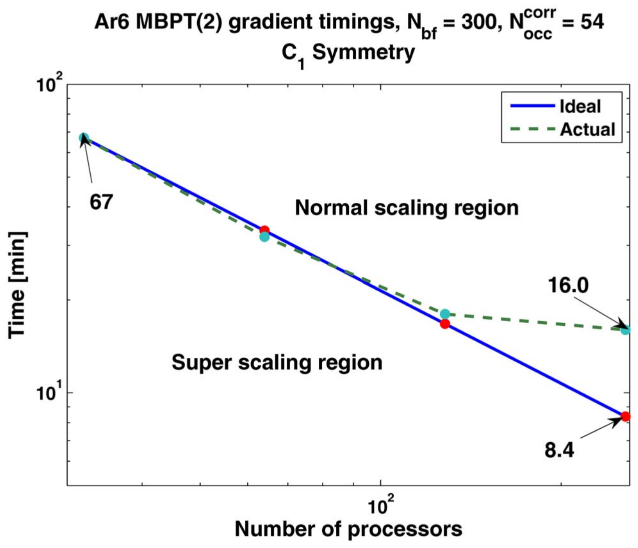

FIG. 3. (Color online) Times for UHF MBPT(2) gradient of Ar_6 cluster (54 correlated occupied electrons, 300 bf) calculations for 32, 64, 128, and 256 processors.

TABLE III. Time in minutes to perform one CCSD gradient calculation for the Ar_6 molecule on the ARSC Opteron cluster. A UHF reference was used, the basis was aug-cc-pVTZ and the symmetry C_1. The number of basis functions used was 300 and the number of correlated occupied orbitals was 54. The number of workers/server was 3/1. CCSD and Lambda times are per iteration. The time to compute the perturbative triples contribution to the energy was with the core dropped.

N_p	Module	t	t^{ideal}	t^{ideal}/t
32	CCSD	103.5	103.5	1.00
	Lambda	119.7	119.7	1.00
	Grad	1273	1273	1.00
	Total	3505	3505	1.00
	CCSD(T)	784	784	1.00
64	CCSD	60.2	51.8	0.86
	Lambda	57.9	59.9	1.03
	Grad	515	637	1.24
	Total	1696	1753	1.02
	CCSD(T)	312	392	1.26
128	CCSD	24.0	25.9	1.08
	Lambda	31.8	29.9	0.94
	Grad	258	318	1.23
	Total	816	876	1.07
	CCSD(T)	178	196	1.10
256	CCSD	13.6	12.9	0.95
	Lambda	16.0	15.0	0.94
	Grad	141	159	1.13
	Total	437	438	0.98
	CCSD(T)	131	98	0.75

The port to Linux clusters with InfiniBand was relatively simple after we had the pure MPI version. We tried the OpenMPI version first because it claims to be thread safe, albeit largely untested. However, we found it to be not functional in a true multithreaded environment. OpenMPI works fine for the pure MPI version of the SIP. The port to the Cray XT3 and IBM Blue Gene architectures also posed no problems, but these have not been optimized yet.

V. PERFORMANCE RESULTS

In this section, we show the results of some calculations done with ACES III on a set of molecules, some of which have been used by others to exhibit the performance of their parallel implementations. We try to carefully state exactly when special algorithms are used to reduce the total amount of data processed or the total number of floating point operations performed. We show wall-clock timings and for some systems we show scaling with number of processors. Depending on the size of the computational problem, it is not feasible to run some calculations on a single processor, or even on a small number of processors. Thus, our scaling results are for different ranges of numbers of processors.

TABLE IV. Time in minutes to perform one CCSD energy calculation for the Ar_6 molecule on the NCSA Altix system. A RHF reference and the aug-cc-pVTZ basis with C_1 symmetry were used. The number of basis functions used was 300 and the number of correlated occupied orbitals was 54. The number of workers/server was 3/1. CCSD times are per iteration. The last column show the ration of the time on the Altix with RHF reference over the time on the Sun Opteron with UHF reference from Table III.

N_p	t	t^{ideal}	t^{ideal}/t	t^{UHF}/t
32	22.2	22.2	1.00	4.66
64	10.9	11.1	1.02	5.52
128	5.9	5.6	1.05	4.07

A. Argon cluster

First, we consider a computation for several representative methods on a small cluster of argon. The advantage of these clusters is that they present no problems with convergence of any kind: The iterative solution of the SCF, coupled cluster, and Λ equations poses no problems and completes in a small number of iterations. These clusters also allow one to increase the size of the problem in a controlled manner to provide sufficient amounts of work when measuring performance on computer systems with more processors.

We choose a cluster with six atoms to make sure that all computations from 32 processors up to 256 processors can complete within the standard queue limits on the Sun cluster with AMD Opteron CPUs and InfiniBand interconnect at Arctc Regional Supercomputer Center (ARSC). We chose the aug-cc-pVTZ basis which puts 50 basis functions on every atom, a total of 300 basis functions. For this calculation all electrons were correlated, which leads to 54 correlated occupied orbitals for each spin. Our calculations use a UHF reference and C_1 symmetry. In Table II and Fig. 3 we show the results for the MBPT(2) gradient. Observe that scaling is good up to 128 processors, but drops off significantly for 256 processors. This is a sign that there is not enough work in this problem to keep 256 processors busy during communication delays.

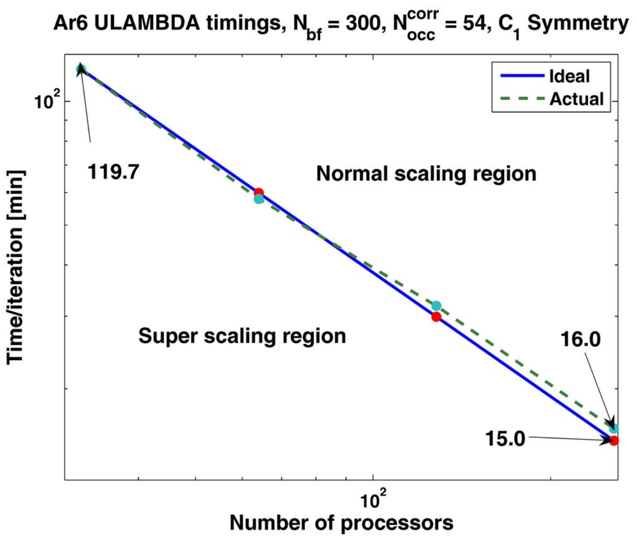

FIG. 5. (Color online) Scaling of UHF CCSD Λ equation of Ar_6 cluster (54 electrons, 300 bf) calculations for 32, 64, 128, and 256 processors.

The MBPT(2) Hessian was computed using a RHF reference, C_1 symmetry, and the same aug-cc-pVTZ basis on 128 processors in the ARSC Opteron cluster. The total calculation took 36987 s. The Hessian has $324=18^2$ elements, of which 171 are unique. The algorithm we implemented computes all 324 elements in a way that is not symmetric, but it significantly faster than a manifestly symmetric algorithm that computes the 171 unique elements. The computation for each Hessian element has two major parts, one involving the second derivatives of the two-electron integrals, the other involving two sets of transformed, differently, first derivatives of the two-electron integrals. For Hessian element, the derivative-integral computation with respect to one coordinate and one kind of transformation takes 155 seconds, the computation and transformation for the other coordinate takes 330 s and the computation with these of the contribution to the Hessian element takes 16 s.

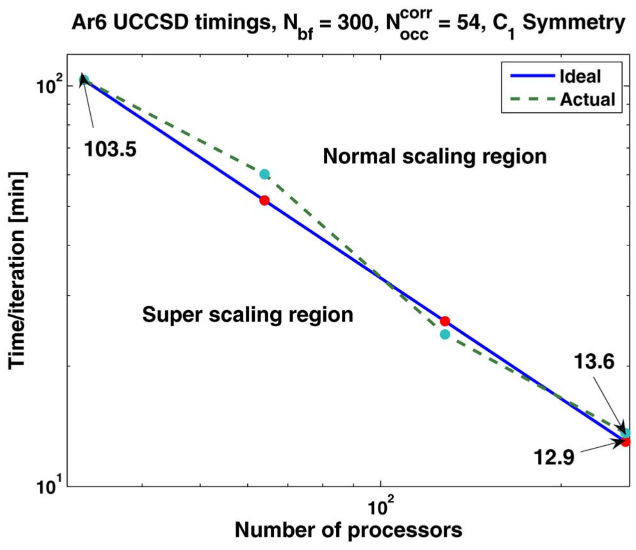

FIG. 4. (Color online) Scaling of UHF CCSD energy of Ar_6 cluster (54 electrons, 300 bf) calculations for 16, 32, 64, and 128 processors.

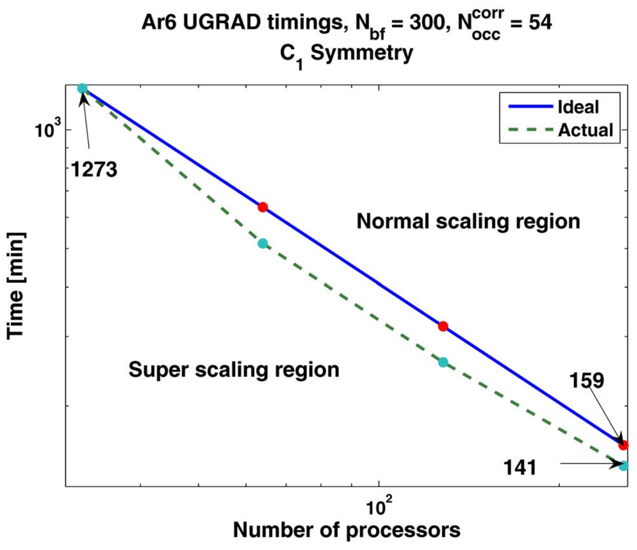

FIG. 6. (Color online) Scaling of UHF CCSD gradient step of Ar_6 cluster (54 electrons, 300 bf) calculations for 32, 64, 128, and 256 processors.

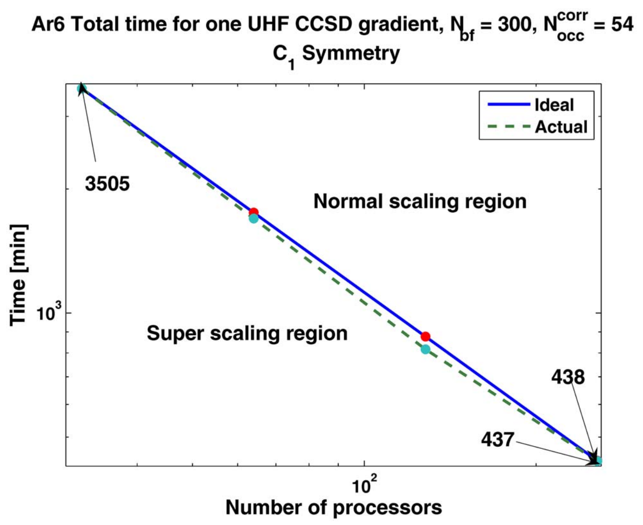

FIG. 7. (Color online) Scaling of one complete UHF CCSD gradient of Ar_6 cluster (54 electrons, 300 bf) calculations for 32, 64, 128, and 256 processors.

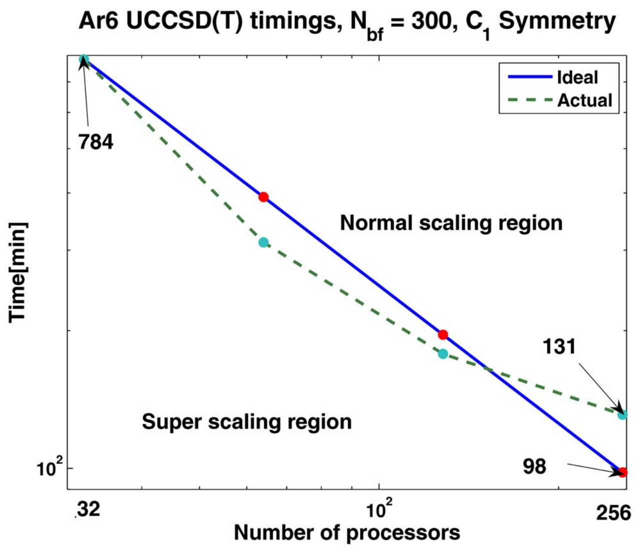

FIG. 8. (Color online) Scaling of UHF CCSD(T) dropped core energy of Ar_6 cluster (54 electrons, 300 bf) calculations for 32, 64, 128, and 256 processors.

The next example is the computation of a CCSD gradient with a UHF reference and C_1 symmetry and the same basis. In Table III, we summarize the results obtained on the ARSC Opteron cluster. We show the times for the individual steps to complete one gradient calculation, which are CCSD amplitudes (and energy), Λ amplitudes, and the gradient construction.

The performance can depend noticeably on the computer system as can be seen in Table IV. We show the CCSD energy calculation with the same basis and symmetry but with a RHF reference for 32, 64, and 128 processors. The amount of work for RHF is about three times less than for UHF. However, the Altix is between four and five times faster than the Sun Opteron cluster. We expect this to be caused by the parallel file system on the Opteron cluster being slow, possibly because of other jobs running on the cluster.

Figures 4–6 show that the individual steps of the gradient calculation scale as well with the number of processors as the total time, as shown in Fig. 7. Observe that good scaling now extends comfortably to 256 processors, unlike the MBPT(2) gradient calculation.

We performed the CCSD(T) energy computation for the Ar_6 cluster with the core orbitals dropped. These results are also shown in Table III. Figure 8 displays the scaling. From the graph, on ecan observe that this calculation is optimally efficient on 64 and 128 processors on the Opteron cluster with InfiniBand interconnect at ARSC.

B. Luciferin

The luciferin molecule has been used before to show the performance of new methods and implementations.[24,29,36] We use an aug-cc-PVDZ basis, which results in 498 basis functions for this molecule, a RHF reference and we correlate 46 orbitals in our CCSD calculation, matching the parameters of the MBPT(2) calculation reported by Baker and Pulay[1] and by Ishimura *et al.*[36] The results for the CCSD energy calculation are shown in Table V and Fig. 9. The scaling is good from 32 to 256 processors.

In Fig. 9, the results for the CCSD(T) energy calculation with 128 processors are also shown. All 46 occupied orbitals are correlated, also in the (T) step. This 498 basis function calculation can be compared to the 160 basis function CCSD(T) calculation reported by Bentz *et al.*[29]

C. Sucrose

We also performed a calculation of the CCSD energy on sucrose, using the same 6-311G** basis set with 546 functions as was used for an MBPT(2) calculation by Janowski *et al.*[37] Our calculation also uses a RHF reference and 68 occupied orbitals are correlated. Table VI lists the results using 32 to 512 processors.

The results are graphically displayed in Fig. 10. Figure 10 shows that the scaling for 256 and 512 processors is very good, but scaling drops off for 128, 64, and 32 processors

TABLE V. Time per iteration to perform the CCSD calculation on the luciferin molecule ($C_11H_8O_3S_2N_2$). A RHF reference was used with the aug-cc-pVDZ basis, which has 498 functions for this molecule. The number of correlated occupied orbitals was 46. The division of processors into workers and servers is also shown.

N_p	N_w	N_s	t_{iter}	$t_{\text{iter}}^{\text{ideal}}$	$t_{\text{iter}}^{\text{ideal}}/t_{\text{iter}}$
32	24	8	115.9	115.9	1.00
64	48	16	46.0	58.0	1.26
128	96	32	22.7	29.0	1.28
256	192	64	13.1	14.5	1.11

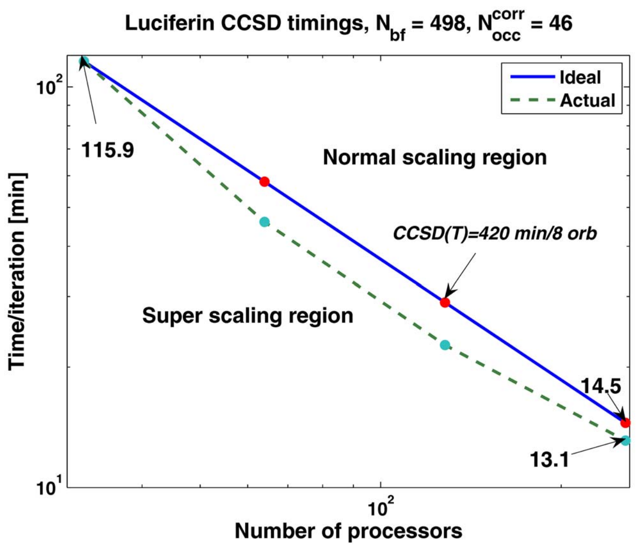

FIG. 9. (Color online) Timing in minutes of RHF CCSD energy of luciferin $C_{11}H_8O_3S_2N_2$ in the aug-cc-pVDZ basis (46 occupied orbitals, 498 bf) calculations on 32, 64, 128, and 256 processors. The time for the CCSD(T) calculation for 8 occupied orbitals with 128 processors is also shown.

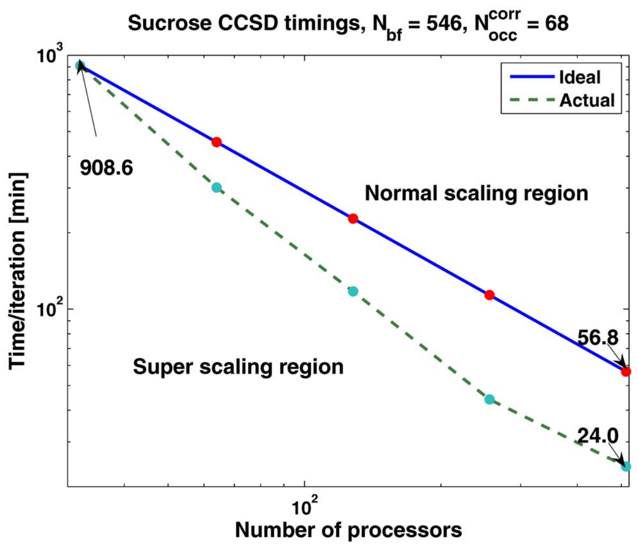

FIG. 10. (Color online) Scaling of RHF CCSD energy of sucrose $C_{12}H_{22}O_{11}$ 6−311G^{**} basis (68 occupied orbitals, 546 bf) calculations for 16, 32, 64, 128, 256, and 512 processors. Linear scaling is ideal.

runs. This is the result of the reference calculation on 32 processors being comparatively slow: the amount of data requires the use of served, i.e., disk resident, arrays and with only 32 processors the input-output delays cannot be effectively masked behind computation by the SIP runtime environment. The calculations on luciferin and sucrose show that the user faced with performing large computations must consider the balance between three ingredients.

(1) The size of the problem in electronic structure theory is determined by the molecule, the method, and the basis set size.
(2) The software and algorthims used.
(3) The computer hardware and the number of processors.

With the the requirement that the computation be done within a useful amount of time from a human point of view, from a number of hours up to one or two days, and the problem given, the two variables at the researcher's disposal are the software and the number of processors. For the luciferin problem on an Opteron cluster with Infiniband interconnect the most cost efficient ACES III run uses 128 processors. For the sucrose problem, the run with 512 processors is the most efficient.

D. Dimethyl-methyl-phosphonate and DMMP+OH

Finally, we present a set of representative results for a molecule that is the object of study in a project of one of the ACES III users. The results of MBPT(2) gradient calculations on dimethyl-methyl-phosphonate (DMMP) (16 atoms and 66 electrons) in a cc-pcVTZ(cc-pVTZ:p) basis are shown in Table VII, with a RHF reference. The results, shown graphically in Fig. 11, clearly scale well up to 128 processors.

In Table VIII, we show the UHF CCSD(T) energy computation timing for DMMP+OH with a smaller basis of 208 basis functions. The calculation was done with 64 processors consisting of 48 workers and 16 servers on the National Center for Supercomputing Applications (NCSA) Altix system "cobalt." In this run, each processor has 1.6 Gbytes, of random access memory assigned to it.

Because the (T) step is naturally parallel, we split the work into seven independent 64-processors jobs, each computing the contributions for a fraction of the occupied orbitals. This is equivalent to running a 448 processor job. However, the realities of scheduling on large shared machines, such as the NCSA Altix 3700 system cobalt, are such that a single job requesting 448 processors may wait in the queue a significant amount of time. We submitted seven independent jobs each requesting 64 processors. These jobs were scheduled almost immediately and ran mostly in parallel. The

TABLE VI. Time per iteration to perform the CCSD calculation on the sucrose molecule ($C_{12}H_{22}O_{11}$). A RHF reference was used and the basis used was 6-311G** with 546 functions. The number of correlated occupied orbitals was 68.

N_p	N_w	N_s	t_{iter}	$t_{\text{iter}}^{\text{ideal}}$	$t_{\text{iter}}^{\text{ideal}}/t_{\text{iter}}$
32	24	8	908.6	908.6	1.00
64	48	16	301.1	454.3	1.51
128	96	32	117.5	227.1	1.93
256	192	64	44.1	113.6	2.58
512	384	128	24.0	56.8	2.37

TABLE VII. Time to perform one MBPT(2) gradient calculation on the DMMP molecule (dimethyl methyl-phosphonate). A RHF reference was used. The basis used was cc-pcVTZ(cc-pVTZ for p), and C_1 symmetry was used. The number of basis functions was 397 and the number of correlated occupied orbitals was 33. The ratio of workers/servers was 3/1 for all computations.

N_p	t	t^{ideal}	t^{ideal}/t
8	689	689	1.00
16	371	345	0.93
32	203	172	0.85
64	95	86	0.91
128	46	43	0.93

longest of these seven jobs took a time that is the same order of magnitude as the CCSD step (6653 s compared to 4119 s). It seems that this is practical way to perform CCSD(T) computations.

Table IX indicates that the time required to solve the UHF LinCCSD equations for DMMP+OH is about half the time required to solve the CCSD equations when comparing computations done on 16 processors. This is consistent with timing data on the CCSD computations indicating that calculation of the quadratic terms is about the same as that of the linear ones. The computation of the CCSD or LinCCSD gradient requires storage of the V_{ci}^{ab} arrays. The calculation of this array is shown in the column marked "transformation," which shows that this partial two-electron integral transformation is rather efficient.

Table X shows that solution of the LinCCSD equations exhibits super linear scaling. The CCSD and LinCCSD algorithms treat all transformed two-electron integrals as served arrays. The CC-amplitudes are also treated as served arrays, as well as the history of five amplitude arrays for the DIIS convergence acceleration method. This represents a significant amount of disk space. For the system considered, DMMP+OH, one two-particle amplitude array is about 0.5 Gbytes and one V_{ci}^{ab} array is about 2.7 Gbytes in size.

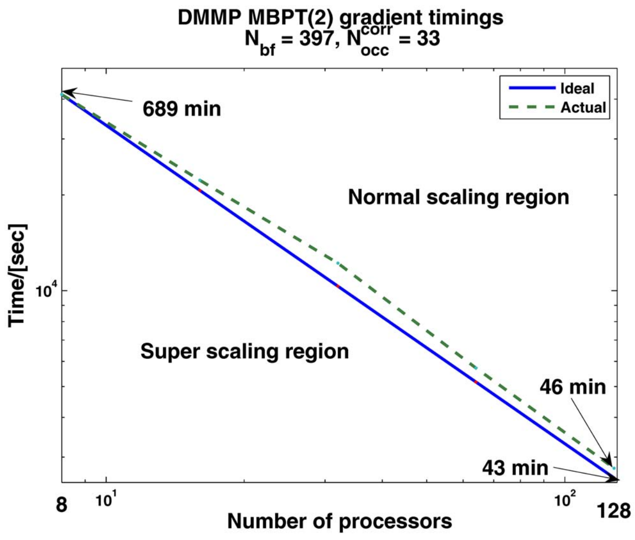

FIG. 11. (Color online) Scaling of RHF MBPT(2) gradient of DMMP $H_{10}C_3O_4P$ (66 electrons, 397 bf) calculations for 8, 16, 32, 64, and 128 processors.

TABLE VIII. Timings for CCSD(T) energies of DMMP+OH (18 atoms, 75 electrons, 208 bf) on the NCSA SGI Altix 3700 system with 64 processors, 48 workers and 16 servers, for the SCF, transformation, and CCSD steps. Times in seconds. For the (T) step, 448 processors were used in the form of seven independent 64-processor jobs.

SCF (64 proc)	Transformation (64 proc)	CCSD (64 proc)	(T) (448 proc)
210	82	4119	6653

The latter is directly treated in the LinCCSD algorithm. Therefore, the amount of disk space needed is significantly less. In all computations, the larger the number of processors being used, and hence, the larger the distributed memory is, the more data can fit into memory. Therefore, less input-output, which is slower, is required as the number of processors increases and the total wall-clock time of the computation can become smaller provided communication between the larger number of processors can be done faster than input-output.

Results of UHF CCSD gradient calculations on DMMP+OH with 208 basis functions can be found in Table XI for 16 to 128 processors of the Army Research Laboratory (ARL) IBM SP4 computer. All 75 electrons are correlated.

The minimum number of processors and memory per processor required to complete the gradient calculation is 32. Scaling results are shown in Fig. 12. The reference time taken from the 16-processor run for the CCSD part of the computation and from the 32-processor run for all other parts. The CCSD and LAMBDA parts exhibit nearly perfect scaling. In fact, the Λ computation on 64 processors shows super scaling of 1.21. The grad part shows super linear scaling on 128 processors, the worst scaling of 0.97 being two-grad on 64 processors.

VI. CONCLUSION

The super instruction architecture has proven to be a very effective tool to produce parallel implementations of several important electronic structure methods. The performance and the scaling of software are competitive with code produced using explicit MPI and OpenMP or POSIX threads programming on a wide range of computer architectures.

The SIP has been instrumented with detailed timing reporting capabilities, thus allowing us to gather detailed statistics from every run. The overhead of this reporting facility has been observed to be negligible. One of the design goals for ACES III was to have software that is flexible for tuning to

TABLE IX. Timings for UHF LinCCSD gradient of DMMP+OH (18 atoms, 75 electrons, 208 bf) runs on different numbers of IBM SP4 processors in seconds.

Module	16	32	64	128
LinCCSD	37 018	12 413	5267	2331
Transformation	1 172	753	533	137
Grad	27 034	11 967	5091	2519

TABLE X. Relative timings (speedup) for the LinCCSD, Transformation, grad contributions to the LinCCSD gradient of DMMP+OH (18 atoms, 75 electrons, 208 bf). Numbers in parenthesis show scaling efficiency.

N_p	LinCCSD	Transformation	Grad	Linear speedup
16	1.00(1.00)	1.00(1.00)	1.00(1.00)	1(1.00)
32	2.98(1.49)	1.56(0.78)	2.26(1.13)	2(1.00)
64	7.03(1.76)	2.20(0.55)	5.31(1.33)	4(1.00)
128	15.87(1.98)	8.81(1.10)	10.73(1.34)	8(1.00)

varying hardware architectures and configurations. It is our experience that this goal was achieved.

One surprising outcome of the project is the dramatic reduction time needed to implement alternative algorithms. Writing SIAL code from theoretical formulas is simpler than writing efficient Fortran code and the ways to get consistently good performance are easily learned and expressed quite simply in the language. Our programmers often implemented several algorithms to compare performance and for debugging.

From the section on performance, the reader can deduce, with some simple checking of the literature, that ACES III is not the fastest parallel program on many of the published examples. However, we have found that ACES III shows very robust performance and scaling on a wide range of molecules and computer systems. This allows practicing theoretical and computational chemists to perform larger calculations with a high degree of predictability on shared resources within real-life limitations of batch queue systems.

As can be seen in Fig. 10, the scaling of ACES III deteriorates for 32 processors, but the program does run correctly and completes in a reasonable time. The super instruction architecture provides a flexible framework where data moves from local to distributed to disk-based with minimal intervention from the programmer. ACES III will run faster if more RAM is available per processor, but it will perform quite respectably with just 1 GB per processor. It will run large problems on 512 processors, but it will also run them, although slower, on 32 processors. This flexibility provides ACES III with robust performannce characteristics required for real-life applications.

We believe that the super instruction architecture includes the best features from many ideas for parallel software design developed over the past decades and implemented in middleware and toolkits like GA,[22,23] DRA,[30] DDI,[29,27] AF,[35] and that it unifies these ideas into a single, easy to program runtime framework. We believe that it is a

TABLE XI. Timings for the UHF CCSD gradient of DMMP+OH (18 atoms, 75 electrons, 208 bf) on different numbers of ARL IBM SP4 processors in seconds.

Module	16	32	64	128
CCSD	64 055	33 139	16 302	8808
LAMBDA	–	35 026	14 467	9088
Grad	–	14 483	6 794	2829

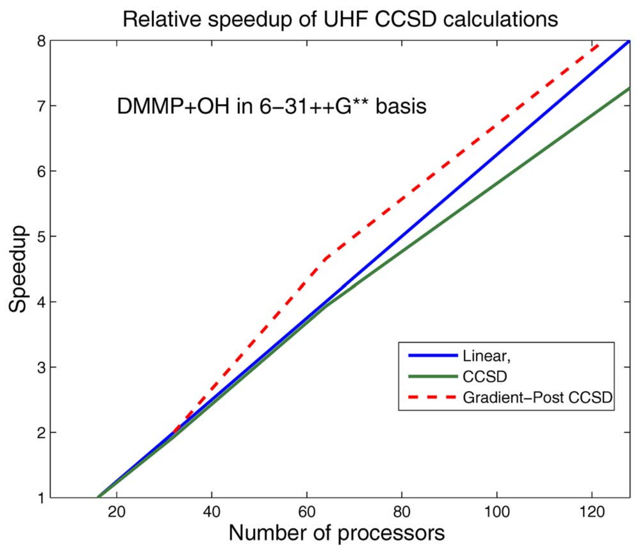

FIG. 12. (Color online) Scaling of UHF CCSD gradient of DMMP+OH (18 atoms, 75 electrons, 208 bf) calculations for 16, 32, 64, and 128 processors.

solid foundation to build software that will allow chemists to routinely and productively tackle much larger chemical problems than is the current standard.

ACKNOWLEDGMENTS

The development of ACES III was largely supported by the US Department of Defense's High Performance Computing Modernization Program (HPCMP) under two complementary programs: Common High Performance Computing Software Initiative (CHSSI), project CBD-03, and User Productivity Enhancement and Technology Transfer (PET). We are grateful to Dr. Sudhakar Parmidiganntham for his strong support and to NCSA and ARSC and ARL for providing computer resources to run some of the examples. We are grateful to the anonymous referee for detailed and thoughtful comments, which led to a substantive revision of the original manuscript and what we consider a much improved paper.

[1] G. D. Purvis and R. J. Bartlett, J. Chem. Phys. **76**, 1910 (1982).
[2] J. D. Watts, J. Gauss, and R. J. Bartlett, J. Chem. Phys. **98**, 8718 (1993).
[3] R. J. Bartlett, *Modern Electronic Structure Theory*, edited by D. R. Yarkony (World Scientific, Singapore, 1995), Vol. II, pp. 1047–1131.
[4] R. J. Bartlett and M. Musial, Rev. Mod. Phys. **79**, 291 (2007).
[5] J. F. Stanton, J. Gauss, J. D. Watts, W. J. Lauderdale, and R. J. Bartlett, Int. J. Quantum Chem. **S26**, 879 (1992).
[6] J. F. Stanton, J. Gauss, J. D. Watts, W. J. Lauderdale, and R. J. Bartlett, ACES II program system, University of Florida, 1994.
[7] J. Gauss, J. F. Stanton, and R. J. Bartlett, J. Chem. Phys. **95**, 2623 (1991).
[8] J. F. Stanton, J. Gauss, J. D. Watts, and R. J. Bartlett, J. Chem. Phys. **94**, 4334 (1991).
[9] R. J. Bartlett and J. D. Watts, *Encyclopedia of Computational Chemistry*, edited by P. von Scheyer *et al.* (Wiley, New York, 1999).
[10] J. F. Stanton, J. Gauss, J. D. Watts, P. G. Szalay, and R. J. Bartlett, ACES II, Mainz–Austin–budapest Version 2005, a quantum chemical program package, University of Mainz, 2005, with contribution from A. A. Auer, D. B. Bernholdt, O. Christiansen, M. E. Harding, M. Heckert, O. Heun, C. Huber, D. Jonsson, J. Juselius, W. J. Lauderdale, T. Metzroth, K. Ruud, and the integral packages: J. Almlf and P. R. Taylor, MOLECULE; P. R. Taylor, PROPS; and T. Helgaker, H. A. A. Jensen, P. Jrgensen, and J. Olsen, ABACUS.
[11] E. A. Salter, G. W. Trucks, and R. J. Bartlett, J. Chem. Phys. **90**, 1752 (1989).
[12] J. Gauss, J. F. Stanton, and R. J. Bartlett, J. Chem. Phys. **95**, 2623 (1991).

[13] R. J. Bartlett, J. D. Watts, S. A. Kucharski, and J. Noga, Chem. Phys. Lett. **165**, 513 (1990).
[14] J. A. Pople, K. Ragavachari, G. W. Trucks, and M. Head-Gordon, Chem. Phys. Lett. **157**, 479 (1989).
[15] P. Pulay, J. Comput. Chem. **3**, 556 (1982).
[16] A. Dalgarno and A. L. Stewart, Proc. R. Soc. London, Ser. A **237**, 245 (1958).
[17] N. C. Handy and H. F. Schaefer III, J. Chem. Phys. **81**, 5031 (1984).
[18] J. F. Stanton, J. Gauss, and R. J. Bartlett, Chem. Phys. Lett. **195**, 194 (1992).
[19] D. E. Bernholdt, Parallel Comput. **26**, 945 (2000).
[20] J. D. Watts, Parallel Comput. **26**, 857 (2000).
[21] A. P. Rendell, T. J. Lee, and A. Komornicki, Chem. Phys. Lett. **178**, 462 (1991).
[22] R. J. Harrison, Theor. Chim. Acta **84**, 363 (1993).
[23] I. Nieplocha, R. J. Harrison, and R. J. Littlefield, in *Proceedings of Supercomputing 1994*, IEEE Computer Society (Washington, D.C., 1994), p. 340.
[24] R. A. Kendall, E. Apra, D. E. Bernholdt, E. J. Bylaska1, M. Dupuis, G. I. Fann, R. J. Harrison, J. Ju, J. A. Nichols, J. Nieplocha, T. P. Straatsma, T. L. Windus, and A. T. Wong, Comput. Phys. Commun. **128**, 268 (2000).
[25] G. Karlströn, R. Lindh, P.-A. Malmqvist, U. Ryde, V. Veryazov, P.-O. Widmark, M. Cossi, B. Schimmelpfennig, P. Neogrady, and L. Seijo, Comput. Mater. Sci. **28**, 222 (2003).
[26] H. Löschka, R. Shepard, R. M. Pitzer, I. Shavitt, M. Dallos, T. Muüller, P. Szalay, M. Seth, G. S. Kedziora, S. Yabushita, and Z. Zhang, Phys. Chem. Chem. Phys. **3**, 664 (2001).
[27] G. D. Fletcher, M. W. Schmidt, B. M. Bode, and M. S. Gordon, Comput. Phys. Commun. **128**, 190 (2000).
[28] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, T. L. Windus, M. Dupuis, and J. A. Montgomery, J. Comput. Chem. **14**, 1347 (1993).
[29] J. L. Bentz, R. M. Olson, M. S. Gordon, M. W. Schmidt, and R. A. Kendall, Comput. Phys. Commun. **176**, 589 (2007).
[30] S. Krishnamoorthy, G. Baumgartner, C.-C. Lam, J. Nieplocha, and P. Sadayappan, J. Supercomput. **36**, 153 (2006).
[31] C. L. Janssen, I. B. Nielsen, M. L. Leininger, E. F. Valeev, and E. T. Seidl, The massively parallel quantum chemistry program (MPQC), Version 2.2, Sandia National Laboratories, Livermore, CA, 2003 (http://www.mpqc.org).
[32] E. F. Valeev and C. L. Janssen, J. Chem. Phys. **121**, 1214 (2004).
[33] I. M. B. Nielsen and J. Curtis L, J. Theor. Comput. Chem. **3**, 71 (2007).
[34] J. Baker and P. Pulay, J. Comput. Chem. **23**, 1150 (2002).
[35] A. R. Ford, T. Janowski, and P. Pulay, J. Comput. Chem. **28**, 1215 (2007).
[36] K. Ishimura, P. Pulay, and S. Nagase, J. Comput. Chem. **27**, 407 (2006).
[37] T. Janowski, A. R. Ford, and P. Pulay, J. Chem. Theory Comput. **3**, 1368 (2007).
[38] R. Bukowski, W. Cencek, K. Patkowski, P. Jankowski, M. Jeziorska, M. Kolaski, and K. Szalewicz, Mol. Phys. **104**, 2241 (2006).
[39] C. Hättig, A. Hellweg, and A. Köhn, Phys. Chem. Chem. Phys. **8**, 1159 (2006).
[40] S. Saebø and P. Pulay, J. Chem. Phys. **115**, 3975 (2001).
[41] T. Daniel Crawford, C. David Sherrill, E. F. Valeev, J. T. Fermann, R. A. King, M. L. Leidinger, S. T. Brown, C. L. Janssen, E. T. Seidl, J. P. Kenny, and W. D. Allen, J. Comput. Chem. **28**, 1610 (2007).
[42] P. Musch and B. Engels, J. Comput. Chem. **27**, 1055 (2006).
[43] N. Flocke and V. Lotrich, J. Comput. Chem. (accepted).