JCTC Journal of Chemical Theory and Computation

Efficient Parallel Implementation of the CCSD External Exchange Operator and the Perturbative Triples (T) Energy Calculation

Tomasz Janowski* and Peter Pulay

Department of Chemistry and Biochemistry, Fulbright College of Arts and Sciences, University of Arkansas, Fayetteville, Arkansas 72701

Received April 24, 2008

Abstract: A new, efficient parallel algorithm is presented for the most expensive step in coupled cluster singles and doubles (CCSD) energy calculations, the external exchange operator (EEO). The new implementation requires much less input/output than our previous algorithm and takes better advantage of integral screening. It is formulated as a series of matrix multiplications. Both the atomic orbital integrals and the corresponding CC coefficients are broken up into smaller blocks to diminish the memory requirement. Integrals are presorted to make their sparsity pattern more regular. This allows the simultaneous use of two normally conflicting techniques for speeding up the CCSD procedure: the use of highly efficient dense matrix multiplication routines and the efficient utilization of sparsity. We also describe an efficient parallel implementation of the perturbative triples correction to CCSD and related methods. Using the Array Files tool for distributed filesystems, parallelization is straightforward and does not compromise efficiency. Representative timings are shown for calculations with 282−1528 atomic orbitals, 68−228 correlated electrons, and various symmetries, C_1 to C_{2h}.

1. Introduction

Since their initial implementations in the late 1970s,[1−3] coupled cluster (CC) methods[4,5] have became one of the most successful tools of quantum chemistry. Although the CC equations are more complicated than the configuration interaction (CI) equations, the benefits they offer over CI methods are significant. In particular CC methods are size consistent (size extensive). This is very important when calculating interaction energies, particularly for weak interactions, or reaction enthalpies. The CC method with single and double substitutions (CCSD) still has significant residual errors. However, when triple substitutions are included perturbatively,[6−8] the accuracy of CC becomes almost quantitative, provided that large basis sets are used and the wave function is dominated by a single configuration. Both the size consistency and the improved accuracy relative to a truncated CI expansion are caused by the implicit inclusion of all higher order substitutions that are products of those of lower order. The quadratic CI (QCI) method is a simplified version of the CC method;[9] in QCISD, some higher order terms involving single substitutions are omitted.

We have implemented the calculation of CCSD energies in parallel for a closed-shell reference in the PQS program package.[10,11] Recently, we added the most widely used perturbative triples correction (T),[7] allowing the calculation of CCSD(T) energies. Several related many-body methods which can be considered as simplified versions of the full CC method have also been implemented: quadratic CI (QCISD), the coupled electron pair approximation (CEPA0 CEPA2),[12] perturbative methods (e.g., MP3, MP4), and variational singles and doubles CI (CISD). The SD part (e.g., CCSD) can use either canonical or noncanonical (e.g., localized) orbitals. The canonical and localized formulations are virtually identical for CI, CCSD, QCISD, etc. However, perturbational methods become iterative in an orbital-invariant, noncanonical form.[13]

The purpose of this paper is to evaluate competing strategies in CC calculation and describe our recent improvements. As our program is able to perform very large calculations on modest hardware, we think it would be

* Corresponding author. E-mail: janowski@uark.edu.

interesting to present details of our current implementation with emphasis on recent improvements and additions. In particular, our method of utilizing sparsity in the external exchange (four virtual indices) part may be of interest, as it combines two techniques that are generally incompatible: the elimination of very small integrals while using high speed dense matrix routines.

Our target hardware is moderately sized PC and workstation clusters.

At the start of this project, the only available distributed memory parallel CC programs were MOLPRO[14] and a developmental version of NWChem.[15] This was surprising because the massive CPU demand and relative simplicity of CC methods makes them ideally suited to parallel processing. More recently ACES II[16] and GAMESS-US[17] joined the list. Very recently, a novel implementation in ACES III was reported.[18] The first parallel CC codes[19−23] were generally developed for the Cray supercomputers and do not perform optimally (if at all) on modern distributed memory workstation clusters.

2. General Remarks about Efficiency

Because of the compute-intensive nature of CC calculations, it is important to achieve the highest possible efficiency. One method of doing this is to rewrite the algorithm in such a way that the compiler can generate optimum code; for instance, by changing the ordering of loops to minimize memory access. Vectorization, known mostly for the Cray series of supercomputers, was a notable example. However, optimization of this kind depends sensitively on the hardware architecture, in particular on memory hierarchy, and may become obsolete when new architectures are introduced. For CCSD(T), a simpler and more general method is to formulate the algorithm as much as possible in terms of matrix operations, in particular matrix multiplications, and use libraries optimized for the particular hardware architecture. Matrix multiplication can be formulated for essentially all computer architectures to run at close to the theoretical maximum speed.[24] We will refer to this strategy as vectorization, despite the fact that it does not conform fully to the original meaning of this term. Our implementation is based on the efficient matrix formulation of the singles and doubles correlation problem, the self-consistent electron pair theory.[25] We use the generator state spin adaptation.[26] This reduces the flop count by a factor of 2 for the pair coupling terms compared to orthogonal spin adaptation and significantly simplifies the formulas. For instance, the CCD residuum formula is only a few lines long. In the SCEP/generator state formulation, parts of the algorithm are already expressed as matrix products and thus run at close to maximum efficiency. Other parts involve linear combinations of matrices or matrix traces. These are "DAXPY" (double precision $aX + Y$) and dot product (DDOT) operations, respectively, which run at a much lower speed because they cannot reuse the fastest (cache) memory efficiently and are therefore limited by much slower main memory access. For instance, the 3 GHz Intel Nocona processor runs a 1000×1000 matrix multiplication at 5128 Mflops/s (using the Goto library)[24] while its performance for a 1 000 000 long dot product is only 465 Mflops/s and for a DAXPY operation with the same flop count is 307 Mflops/s, over 15 times slower than matrix multiplication. We note that it is unlikely that these ratios would change with newer memory architectures, as access to a small (cache) memory is inherently faster than to a large (main) one. It is therefore imperative to reformulate all operation to run as matrix multiplications. For instance, the contribution of four virtual orbitals (the external exchange operator or particle−particle ladder), discussed in the next section, is usually the computationally most demanding part of a CCSD calculation, and its natural formulation does not take the form of a matrix multiplication.

3. External Exchange Operator

This is usually the most demanding part of CCSD/QCISD programs. Possible methods of calculation include the atomic orbital (AO) or molecular orbital (MO) forms. If integral prescreening is not used, or does not yield significant savings, e.g., for small molecules, the MO formulation is faster, as the number of virtual MOs is always smaller than the total number of basis functions (AO). This is significantly magnified by the fourth power, giving a factor of 2 for systems where the number of virtual orbitals is only 16% smaller than the number of AOs. However, the AO formulation allows integral direct implementations, avoiding storage of four-external integrals and enabling utilization of sparsity. As with increasing computer power, we are able to treat bigger systems and sparsity becomes increasingly important, yielding 90% savings for one of our test molecules descibed later in this paper. On the other hand, contemporary disk capacities have grown significantly, so it is not obvious that integral direct methods are more advantageous than storing integrals on disk. Parallel filesystems, such as our Array Files tool[27] can use efficiently the aggregate local disk capacity of a cluster, allowing the storage of large integral files.

Nonetheless, the AO formulation, direct or not, allows sparsity utilization, which seems to be unavoidable if we really want big coupled cluster calculations. Note that the importance of sparsity arises from the reduction of the formal operation count of the EEO evaluation, not just from the reduction of integral calculation. While the cost of the integral evaluation is usually smaller than the evaluation of the external exchange, this is not true for small systems using large, diffuse basis sets. In view of the large increase in disk capacities, nondirect AO algorithms may be a viable option.

The EEO is naturally expressed as a dot product:

$$(\mathbf{K}[\mathbf{T}^{ij}])_{\mu\lambda} = \sum_{\nu\sigma} (\mu\nu|\lambda\sigma)\mathbf{T}^{ij}_{\nu\sigma} \tag{1}$$

Here, the Greek letters denote contracted basis functions, and i and j denote molecular orbitals, $\mathbf{T}^{ij}$ is a matrix of amplitudes associated with excitation from the ij orbital pair. If sparsity or symmetry is not used, the operations count is $^1/_2 n^2 N^4$, where n is the number of occupied orbitals and N is the number of AOs. This is the most expensive part of a CCSD calculation, especially if the basis set (N) is large. The latter is usually the case, as CCSD(T) calculations require basis sets of at least triple-ζ or quadruple-ζ quality[28] in order to obtain quantitative results. To take maximum

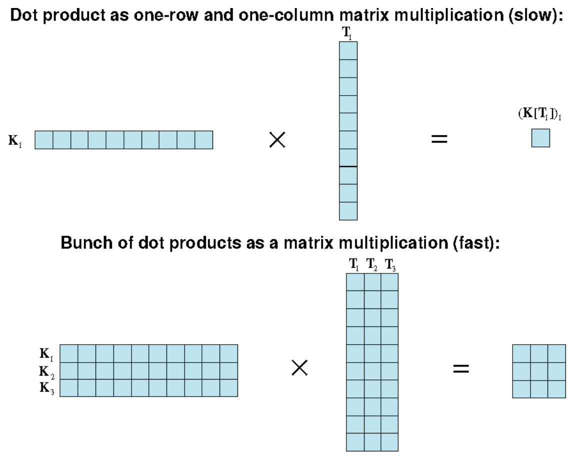
Dot product as one-row and one-column matrix multiplication (slow):
T_1
K_1
$(K[T_1])_1$
Bunch of dot products as a matrix multiplication (fast):
T_1 T_2 T_3
K_1
K_2
K_3

Figure 1. Rearrangement of several dot products (matrix traces) into one big matrix product.

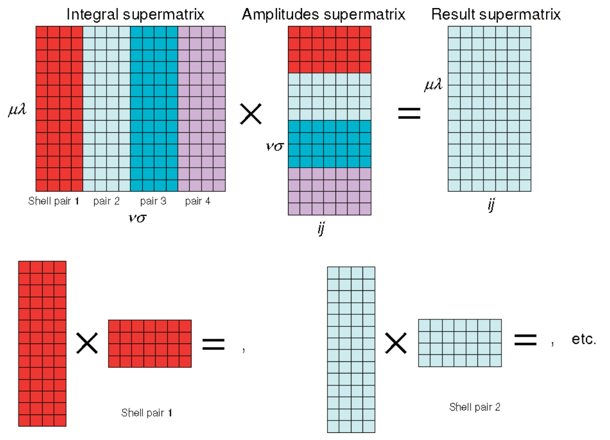

Figure 2. New EEO blocking scheme.

advantage of the efficiency of modern processors, the dot product shown above was reformulated as a matrix product. Figure 1 illustrates how this was done. In practice, we use symmetric and antisymmetric combinations of amplitudes and integrals as described before:[29]

$$(\mathbf{K}[\mathbf{T}^{ij}])^{\pm}_{\mu\lambda} = \frac{1}{4}\sum_{\nu\geq\sigma}[(\mu\nu|\lambda\sigma) \pm (\mu\sigma|\lambda\nu)](\mathbf{T}^{ij}_{\nu\sigma} \pm \mathbf{T}^{ij}_{\sigma\nu})(2-\delta_{\nu\sigma}) \tag{2}$$

The quantity $(\mathbf{K}[\mathbf{T}^{ij}])^{\pm}_{\mu\lambda}$, calculated for $\mu \geq \lambda$ only, gives the final external exchange by taking appropriate linear combinations:

$$\mathbf{K}[\mathbf{T}^{ij}])_{\mu\lambda} = (\mathbf{K}[\mathbf{T}^{ij}])^{+}_{\mu\lambda} + (\mathbf{K}[\mathbf{T}^{ij}])^{-}_{\mu\lambda} \tag{3}$$

and

$$\mathbf{K}[\mathbf{T}^{ij}])_{\lambda\mu} = (\mathbf{K}[\mathbf{T}^{ij}])^{+}_{\mu\lambda} - (\mathbf{K}[\mathbf{T}^{ij}])^{-}_{\mu\lambda} \tag{4}$$

This procedure reduces the formal operation count by a factor of 2, giving $^1/_4 n^2N^4$, significantly better (up to a factor of 5) than in spin−orbital formulations (e.g., $^5/_4 n^2N^4$ in Hirata's work).[30] An undesirable side effect of this technique is that the sparsity of integrals deteriorates somewhat.

Our previous EEO implementation was constrained by the integral program, which generated batches of full **K** matrices, as defined by $(\mathbf{K}^{\mu\lambda})_{\nu\sigma} = (\mu\nu|\lambda\sigma)$, for a given atomic orbital (AO) shell pair $\mu\lambda$. We stored as many **K** matrices as possible, using shell merging techniques, mentioned earlier by Schütz et al.,[31] constructed supermatrices out of them (see Figure 1), then read all the **T** matrices (in batches), constructed a similar supermatrix, and multiplied the two together using matrix multiplication algorithms. This technique was efficient for systems with relatively small basis sets but deteriorated significantly for systems with more than 1000 basis functions where a single matrix can occupy a large amount of memory. A good example is one of our largest calculations performed so far, benzene dimer QCISD or CCSD in the aug-cc-pVQZ basis set (1512 basis functions)[11] (see also ref 32 for more recent QCISD(T) results), where a single matrix occupies about 18 MB of memory. Because of this, we could process only batches of 100−200 $\mu\lambda$ pairs at a time. Each batch required reading the full set of amplitudes from a disk, resulting in excessive input/output (I/O). The amplitudes had to be read 5000−10 000 times for the benzene dimer calculation. The set of all amplitudes for this system occupied about 8 GB, so one full EEO evaluation required the reading of 40−80 TB of data. The speed of the calculation was thus limited by I/O.

We have rearranged this algorithm using an older and somewhat slower but more flexible integral program. This program is not vectorized and therefore can generate integrals for a single quartet of AO shells. This makes it possible to subdivide the range of AO index pairs in smaller blocks, alleviating the memory problem, and in turn dramatically reducing the amount of I/O. Blocking is widely used in high-performance computers, for instance in efficient matrix multiplication routines. In the context of CC methods, the tensor contraction engine of Hirata[30] uses blocks (called "tiles"). However, this technique does not exploit sparsity. A local coupled cluster implementation was reported recently[33] that exploits integral sparsity using an approach different from ours.

We define batches of indices $\mu\lambda$, $\nu\sigma$, and ij. The amplitude matrices are presorted into blocks corresponding to these batches. The outermost loop is parallelized; it runs over batches of $\mu\lambda$. The next loop runs over batches of $\nu\sigma$. At this point, all integrals $(\mu\nu|\lambda\sigma)$ for the $\mu\lambda$ and $\nu\sigma$ batches are calculated. The innermost loop runs over batches of ij. A block of the amplitudes corresponding to ij and $\nu\sigma$ is read from distributed storage, and a block of partial results are calculated by matrix multiplication of $\mathbf{T}^{ij}_{\nu\sigma}$ with $(\mu\nu|\lambda\sigma)$, using $\nu\sigma$ as the summation index. The result is accumulated in $\mathbf{K}[\mathbf{T}^{ij}]_{\mu\lambda}$. After the ij and $\nu\sigma$ loops are finished, the completed $\mathbf{K}[\mathbf{T}^{ij}]_{\mu\lambda}$ matrices are written to distributed disk storage. The dominant memory requirement is for the storage of the $\mathbf{K}[\mathbf{T}^{ij}]_{\mu\lambda}$ matrices and requires only $Bn^2/2$ memory locations where B is the $\mu\lambda$ batch size. Only one full read of the amplitude blocks per slave is needed, a dramatic reduction compared to our original code. e.g., using the example above, with 30 slaves, the full amplitude set is read only 30 times, not 10 000 times. The algorithm is shown in Figure 2.

One may note that, as we calculate all integrals for $\mu \geq \lambda$, only one permutational symmetry out of four is used. This requires extra CPU time compared to algorithms that use the full permutational symmetry, like that of Schütz et al.[31]

However, the advantage of avoiding multiple writings and readings of partial results greatly outweighs the increased cost of integral evaluation, even in MP2. In our opinion, integral permutational symmetry is only of minor importance in correlated calculations, as integral evaluation scales formally only as $O(N^4)$, while EEO for instance scales as n^2N^4. This is underscored by the fact that the main use of integral screening described below is not to economize on integral evaluation but to speed up the matrix multiplications.

The algorithm briefly described above facilitates the implementation of integral screening. For large molecules, the integral matrix contains large numbers of vanishingly small integrals. For example, for a linear decapeptide of ten glycine molecules, about 90% of the integrals are numerically zero, even with a very strict threshold of 10^{-15}. This results in the elimination of many rows (Figure 2) from the left-hand side matrix, allowing compression and the use of efficient dense matrix multiplication routines for the resulting smaller matrices.[11] One technical problem associated with the utilization of screening is that if a larger numbers of $\nu\sigma$ pairs are treated simultaneously, the location of zero integrals becomes more-or-less random and only a few rows can be eliminated. Dividing the matrix into narrower stripes, say over single shell pairs, improves the utilization of sparsity but reduces the efficiency of matrix multiplication and results in more memory copying (e.g., the accumulation of partial results in the final matrix). A compromise must be sought which allows us to treat as many $\nu\sigma$ pairs as possible at a time while still allowing efficient utilization of integral sparsity.[11]

Row sparsity can be improved significantly if we order the $\nu\sigma$ AO index pairs appropriately. It can be theoretically justified that sorting them according to the physical space location increases regularity in the sparsity pattern of the integral matrix. This phenomenon arises from the fact that most integrals are numerically zero if, for a given $(\mu\nu|\lambda\sigma)$ integral, μ is far away from ν or λ is far away from σ. The integral will also be small if the $\mu\nu$ pair is far away from the $\lambda\sigma$ pair, but the asymptotic behavior is different. In the former two cases, the integral decreases exponentially as a function of the distance; in the latter, it is a much weaker $1/r$ dependency. Because one row of our integral matrix (Figure 2) contains all integrals for a given $\mu\lambda$ index, it is better to group all integrals so that the ν and σ atomic orbitals are distant from μ and λ, respectively. Unfortunately, all rows must share the same ordering pattern—otherwise we cannot use matrix multiplication. The best compromise is to group the $\nu\sigma$ pairs according to the coordinates of their centers. We divide the molecule into boxes 2−3 Å big and collect all $\nu\sigma$ AOs that belong to the same pair of boxes. This improves the sparsity for larger batches of $\nu\sigma$ pairs significantly. For instance, if a batch contains ν's from box A and μ is distant from A then the whole $\mu\lambda$ row of the integral matrix $\mathbf{K}_{\nu\sigma}^{\mu\lambda}$ vanishes. The same statement holds true if box B containing σ's is distant from λ.

Our algorithm for sorting shell pairs is as follows. First we divide the AO set into two categories according to their exponent values. One set contains all AOs with large exponents (tight basis functions), the second one, all remaining AOs. This way we can take advantage of the better sparsity of the more compact basis functions and isolate the diffuse ones. Further subdivision, e.g., very tight functions, moderately diffuse, and very diffuse AOs, is also possible. The next two loops are over boxes. Consecutive boxes are adjacent, which improves the sparsity pattern. The inner two loops sort the $\nu\sigma$ pairs inside the boxes according to center locations (atoms). The outcome of the sorting routine is an array with the pairs sorted and additional arrays which indicate the box and atom borders and the number of pairs contained in pairs of atoms and pairs of boxes. In the next step, these arrays are used to determine the $\nu\sigma$ batches. Usually we want to cut the $\nu\sigma$ set either at the shell pair, or better, atom, or, even better, box boundaries. Our goal is to have as large batches as possible, within the memory allocation limits. Large batches increase matrix multiplication efficiency and I/O throughput. Taking all the above into account, the "cut" and batch creation takes place if the current batch is big enough or if a forthcoming batch can be large. The cuts are allowed on the atom or box borders only, with the latter preferred.

In this way, we can utilize at least 50% of the maximum sparsity by eliminating empty rows. The average size of a $\nu\sigma$ stripe is about 100−200, depending on the system and the basis set used. The row sparsity is about the same as if we divided the matrix by shell pairs, which yields much smaller $\nu\sigma$ stripes (it can potentially be just 1 for an ss pair, 3 for an sp pair, etc.).

Parallelization of the EEO part follows the scheme described previously.[11] With Array Files as the I/O software,[27] all nodes share uniform access to network files, i.e., they can access the same data using a given file and record number. Parallelization is performed over batches of $\mu\lambda$ pairs. If the available memory allows, we divide all $\mu\lambda$ pairs into as many batches as the number of slaves. If there is insufficient memory for this, the number of batches is chosen as a multiple of the number of slaves. This guarantees good load balancing. Every slave processes its own batch in essentially the same way as in the single processor program, but uses Array Files I/O routines in place of local I/O. All data are read or written via network communication. We do not use any local caching of the results or amplitudes, as local disk I/O rates are commensurate with network rates.

4. Perturbative Triples Implementation and Parallelization

The key part of the perturbative triples correction to the CCSD energy is the calculation of the following quantity (see e.g.refs 23, 36 or 37):

$$W_{ijk}^{abc} = P_{ijk}^{abc}\left(\sum_d \mathbf{T}_{ij}^{ad}(ck|bd) - \sum_l \mathbf{T}_{il}^{ab}(ck|lj)\right) \qquad (5)$$

Here P is the permutation operator, which performs simultaneous permutation of indices ijk and abc, generating a sum of six terms, each of the above form. The indices $ijkl$ denote Hartree−Fock occupied canonical molecular orbitals, $abcd$, the canonical virtuals.

W is combined into the (T) energy:

$$E(T) = \sum_{a \geq b \geq c} (2 - \delta_{ab} - \delta_{bc}) \sum_{ijk} W_{ijk}^{abc} (4W_{ijk}^{abc} + W_{kij}^{abc} + W_{jki}^{abc} - 2W_{kji}^{abc} - 2W_{ikj}^{abc} - 2W_{jik}^{abc}) / D_{ijk}^{abc} \quad (6)$$

Here D is a standard perturbational energy denominator. Equation 5 requires about $C(n^4V^3 + n^3V^4)$ floating point operations, but eq 6 needs only En^3V^3 (C and E are constants, V is the number of virtual orbitals). For simplicity, we have omitted contributions from singles amplitudes in eq 6, which corresponds to the [T] correction of Urban et al.[8] Singles do not add significantly to the numerical effort, and therefore, our discussion will focus on the doubles part only. We have to calculate and store W^{abc} only for a given *abc* triplet and all *ijk*; after calculating the energy contribution of this set it can be deleted and a new *abc* triplet processed. This requires only n^3 memory locations, allowing the program to calculate a number of *abc* triplets simultaneously. This is advantageous because the integrals and amplitudes in eq 5 can be reused with proper blocking, significantly reducing the I/O overhead.

Equation 6 is often written in an alternative form:[23,36]

$$E(T) = \sum_{i \geq j \geq k} (2 - \delta_{ij} - \delta_{jk}) \sum_{abc} W_{abc}^{ijk} (4W_{abc}^{ijk} + W_{bca}^{ijk} + W_{cab}^{ijk} - 2W_{cba}^{ijk} - 2W_{acb}^{ijk} - 2W_{bac}^{ijk}) / D_{ijk}^{abc} \quad (7)$$

In this form, one has to calculate W^{ijk} for a given *ijk* triplet and all *abc*, requiring V^3 storage. This severely limits the size of systems which can be treated. For example, for 1000 virtual orbitals, we need 8 GB of fast memory to store the partial result. Of course, for a small system, eq 7 vectorizes better, as the matrices are larger. This improves the flop rate for small dimensions, although the effect saturates at a dimension of a few hundred. Notice that the form of energy expression used (eqs 6 or 7) results in different algorithms for the calculation of W (eq 5) and, thus, changes the whole program design.

We have implemented triples using both algorithms. Although the matrix operations as defined by eq 7 are significantly faster for smaller systems, the gain is largely offset by increased I/O. In addition, on our PC cluster (4 GB RAM per node), it limits the calculation to about 500 virtual orbitals. Our final code uses the slower formulation in eq 6, but taking into account memory savings, I/O reduction due to blocking, and ease of implementation, it is more efficient, particularly for big systems. The pseudocode for our algorithm is shown in Figure 3. Our conclusion about the relative merit of eq 6 vs eq 7 is the opposite of what Rendell et al.[23] arrived at using the Cray supercomputers. They argued that eq 7 vectorizes better and that memory will not be a problem in the future. However, CPU speed has increased faster than RAM memory size, making the latter the bottleneck for distributed memory workstations.

In order to explain how W^{abc} is calculated, we need to define a new quantity:

$$X_{ijk}^{abc} = \sum_d \mathbf{T}_{ij}^{ad}(ck|bd) - \sum_l \mathbf{T}_{il}^{ab}(ck|lj) \quad (8)$$

This allows us to cast eq 5 in a more explicit form:

$$W_{ijk}^{abc} = X_{ijk}^{abc} + X_{jki}^{bca} + X_{kij}^{cab} + X_{kji}^{cba} + X_{ikj}^{acb} + X_{jik}^{bac} \quad (9)$$

```
Loop over batches of $A$            ($A$ – batch of virtual indices)
  loop over batches of $B \leq A$
    loop over batches of $C \leq B$

      read & store integrals needed for all indices $a \in A, b \in B, c \in C$
      read & store amplitudes needed for all indices $a \in A, b \in B, c \in C$

      loop over all virtuals $a \in A$
        loop over all virtuals $b \in B$
          loop over all virtuals $c \in C$

            if ($a = b$ and $b = c$) exit loop
            calculate $W^{abc}$ (slow)
            add $W^{abc}$ contribution to energy $E$ (fast, extensively cached operation)

          endloop $c$
        endloop $b$
      endloop $a$

    endloop $C$
  endloop $B$
endloop $A$
```

Figure 3. Outline of the main "driver" loop of the (T) contribution.

```
Zero $W^{abc}$ array. An array has indices $i, j, k$.

Call $X$ for $a, b, c$, add result to $W^{abc}$ array
Rearrange $W^{abc}$ array (swap 2nd and 3rd index)

Call $X$ for $a, c, b$, add result to $W^{abc}$ array
Rearrange $W^{abc}$ array (swap 1st and 2nd index)

Call $X$ for $c, a, b$, add result to $W^{abc}$ array
Rearrange $W^{abc}$ array (swap 2nd and 3rd index)

Call $X$ for $c, b, a$, add result to $W^{abc}$ array
Rearrange $W^{abc}$ array (swap 1st and 2nd index)

Call $X$ for $b, c, a$, add result to $W^{abc}$ array
Rearrange $W^{abc}$ array (swap 2nd and 3rd index)

Call $X$ for $b, a, c$, add result to $W^{abc}$ array

Rearrange $W^{abc}$ array (swap 1st and 2nd index, this call
                          recovers original $ijk$ ordering).
```

Figure 4. W^{abc} construction scheme.

It is sufficient to have only one subroutine for the calculation of X. It is called six times for every index triple $a \geq b \geq c$ in eq 9. The calls are interleaved by another subroutine which interchanges elements of W before adding the next component X. Swapping the destination W array, instead of adding X with permuted *ijk* indices simplifies the program code, as we do not need to write six different subroutines to calculate each term in eq 9. Instead, we call the same subroutine and add partial result to W with two indices swapped. All operations on W, including energy contribution calculation are very fast, as the array W is usually fairly small (2 MB for 63 occupied orbitals) and fits in the cache of most processors. With the algorithm based on the formula eq 7, the calculation of the final energy contribution becomes more expensive, because W^{ijk} is large and requires access to main memory, instead of fully cached operations for W^{abc}. The difference in the time of evaluation of the final energy expression can be up to an order of magnitude. Figure 4 shows an outline of the W calculation part of the triples code.

Vectorization inside the X subroutine is explained in Figure 5. The figure shows the first term of eq 8; the same method is applied to the second term. In Figure 5, the $\mathbf{g}^{bc}$ matrix is a matrix of integrals, $(ck|bd) = (\mathbf{g}^{bc})_{dk}$. The W^{abc} array is obtained by a single matrix multiplication.

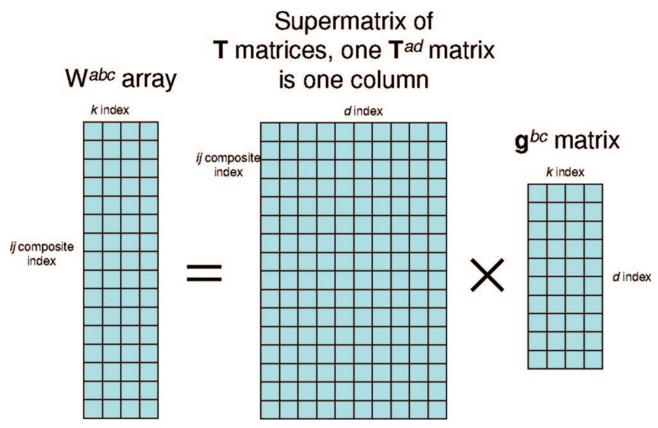

Figure 5. Triples contribution (eq 8, first term), as a matrix product.

The simplicity of accessing global data through the Array Files system allows a simple master-slave self-scheduling parallel algorithm. Slaves ask the master for work and get batches of *ABC* triples to work on. When they are finished they store the partial energy in a local variable and request more work from the master. If the master responds with a "no work" message, the slaves exit their triple substitution subroutine. At the very end a global energy sum is performed over all slaves.

5. Discussion

We describe an improved parallel implementation of CCSD(T) and related methods. The most expensive computational steps, the calculation of the external exchange operators, and the perturbational triples energy are formulated in terms of dense matrix multiplications that take the maximum advantage of modern CPUs. Our new EEO algorithm exploits the sparsity of the atomic orbital integrals, and its communication requirements have been much reduced compared to our previous program. Benchmarks have been presented with over 1500 basis functions and 68 atoms on 2−16 computing nodes.

The main difference between our CC code and the NWChem[15] and GAMESS[17] implementations is that our program makes extensive use of disk storage and, therefore, can perform large calculations on modest size clusters. ACES II[16] uses replicated data structures. This poses limitations on the calculations by the size of the local storage.

Table 2. Perturbative Triples Timings for a Test Set of Molecules

	(nodes/proc) min							
molecule	1/1	2/2	4/4	4/8	8/8	8/16	16/16	16/32
aspirin	1196	601.6	292.5	157	151.3	82	76.7	43
sucrose			12436		5981.9	3216	3034	1594
$(glycine)_{10}$								14319
Si-ring								4168
AMP								16740

Our code shares the general design philosophy of MOLPRO.[14] However, we use Array Files instead of global arrays. This has the merit of enabling small memory workstations to perform large calculations, at the cost of modest overhead. The I/O can be significantly reduced by employing various blocking strategies, as we have discussed already in the EEO and (T) part. The TCE (Tensor Contraction Engine)[30] also employs blocking techniques but does not address sparsity. The TCE can automatically generate code. This restricts it, in its present form, to an orthogonal spin−orbital formulation. As our flop counts show, the hand-coded version achieves substantially better performance, e.g., in the external exchange part, than the automatically generated code. However, automatic code generation is probably unavoidable for higher order CC programs.

The latest CC parallel code, ACES III,[18] uses a novel parallel programming language and automatic blocking of arrays. Preliminary results show very good parallel scaling.

6. Sample Timings

Table 1 presents QCISD timings obtained for a series of molecules of different size. All calculations were performed on a 16-node cluster, connected via gigabit ethernet. Every node of this cluster has a dual core processor (64-bit Intel Pentium IV Prescott, 3.0 GHz), 4 GB of RAM memory, and a 300 GB RAID0 (striped) array of disk scratch storage. Our test set includes a range of molecular sizes and symmetries: aspirin, sucrose, a linear polymer of ten glycine molecules with one symmetry plane, a fragment of a copper-based catalyst with empirical formula $CuAlSi_3O_{12}H_8$, adenosine monophosphate (AMP), and a simple calixarene with empiri-

Table 1. QCISD and Sample CCSD Timings for a Test Set of Molecules

						time per iteration (nodes/proc) min[a]					
molecule[b]	sym	basis	nbf[c]	corr[d]	iter[e]	2/2	4/4	8/8	8/16	16/16	16/32
aspirin	C_1	6-311G**	282	34	16	18.9	11.42	6.4	4.8	3.8	3.1
sucrose	C_1	6-31G**	455	68	12	365.0	197.4	104	61.8	53.7	39.3
$(glycine)_{10}$	C_s	6-31G*	638	114	14	2133	1099.0	590	374	271	203
Si-ring	C_1	VTZ3P[f]	664	53	18	826	443.3	226	127.7	123.5	74
AMP	C_1	def2-tzvp[g]	803	63	14	2384	1166	605.2		325	199
Calix	C_{2h}	cc-pVTZ[h]	1528	92	11[i]			3506		1723	
					sample CCSD timings[j]						
aspirin	C_1	6-311G**	282	34	14	31.7	16.6	10.4	8.6	6.1	5.6
sucrose	C_1	6-31G**	455	68	12	519.2	236.2	123.0	91.2	71.7	65.4

[a] For example, 8/16 means that 16 processes were running on 8 nodes, i.e. 2 processes per node. [b] The test set is described in the Sample Timings section of this paper. [c] Number of basis set functions. [d] Number of correlated orbitals. [e] Number of iterations, until max residue $R < 10^{-6}$ and energy difference $\Delta E < 10^{-6}$. [f] Valence triple-ζ + triple polarization.[38] [g] Valence triple-ζ + polarization.[39] [h] Correlation consistent basis.[40] [i] Only 11 iterations were allowed to complete. [j] Our current integral-direct CCSD implementation requires extra integral recalculation steps, comparing to QCISD.

Table 3. Timings for an External Exchange Evaluation (per Iteration): Comparison of the Old and the New Algorithms

					EEO evaluation time (min)	
molecule	nodes	basis set	no. of basis functions	no. of correlated orbitals	new algorithm	old algorithm
benzene dimer (C_1 symmetry)	32	aug-cc-pVQZ	1512	30	379	1813
(glycine)$_{10}$	16	6-31G*	679	114	72	148

Table 4. Timing for Triples Evaluation in Parallel Displaced Benzene Dimer

molecule	nodes	basis set	no. of basis functions	no. of correlated orbitals	time of triples evaluation	CPU efficiency
benzene dimer (C_{2h} symmetry)	32	aug-cc-pVQZ	1512	1512	3420	93%
as above	32	aug-cc-pVTZ	828	30	350	91%

cal formula $C_{32}H_{32}O_4$ and C_{2h} symmetry. Each process was constrained to use no more than 1.6 GB of RAM.

The parallel scaling of QCISD jobs run with only one process per node is satisfactory, and in some cases, superlinear scaling is observed. This is most likely caused by the fact that the amount of I/O per process decreases with increasing number of processes, allowing the operating system (OS) to cache the data more efficiently in the main memory (in our case, the RAM available for disk caching was about 2.4 GB). Comparing calculations involving the same number of processes but a different number of nodes, Table 1 shows that jobs with two processes per node are significantly slower than jobs which use the same number of processor cores but with one process per node. In the former case, two processes must share access to the same disk. An additional effect arises from the deterioration of OS data caching. In our case two processes occupy 3.2 GB of main memory leaving only about 0.8 GB for the OS to use as I/O cache. Running 32 processes on 16 nodes is still faster than running only 16 processes, but the speedup is modest.

Because the amount of memory available impacts the OS data caching performance, it might seem advantageous to use shared memory parallelization (e.g., multithreading) on the same node, instead of running several separate processes per node. In our previous version of the CCSD program, we have used a multithreaded version of matrix multiplication routines.[24] If the choice is between one process per node without threading and one process per node with threading, the latter is usually (not always) faster. However, the efficiency of the threaded routines depends strongly on the size of matrices being multiplied. For very small (10×10) matrices, it is 1388 Mflop/s on single processor, but only 62 Mflop/s with threading. This probably arises from the overhead of threads preparation. The threaded and single-threaded libraries perform comparably if the matrix dimension is about 50. For large dimensions, the threaded version outperforms the nonthreaded one almost by the theoretical factor. Our current CCSD version uses nonthreaded matrix multiplication libraries for three reasons. (1) In our blocked implementation, the matrices are relatively small, and the threaded libraries do not reach their theoretical limit. (2) Our integral program is not yet thread-safe. (3) Threading does not remove the I/O bottleneck. The latter is diminished in our current algorithm but, as the timings show, not fully eliminated. Note, however, that threading may improve I/O caching by the OS because one threaded process may occupy less memory than two independent processes running concurrently.

A different picture emerges for the calculation of the perturbative triples contribution (Table 2). Triples are very expensive in terms of CPU usage, but the time spent doing I/O is negligible. The I/O reduction results from an efficient blocking algorithm, such that the data are read in batches and reused as frequently as possible (see Figure 3). For large systems, a single process can fill its 1.6 GB of memory storage with data and use it for about 30 min without needing any I/O. This is the reason why the triples contribution scales almost linearly with the number of CPU cores and differences between jobs involving one and two processes per node are small. The calculation of the triples contribution can efficiently utilize both CPU cores because the data are reused more extensively in the triples algorithm than in the SD part.

In Table 3, the efficiency of the old and new EEO algorithms are compared. In order to demonstrate savings and compare the new results with previous ones,[11] we have switched off the use of symmetry for the benzene dimer. The speedup is dramatic, almost a factor of 5. Because the benzene dimer has very little integral sparsity with the aug-cc-pVQZ basis set, the speedup is almost exclusively due to the reduction in I/O.

Table 4 presents sample timings for one of the biggest (T) calculation we have done so far, the benzene dimer with the aug-cc-pVQZ basis set. This calculation was performed on the Red Diamond supercomputer at the University of Arkansas. It has a configuration similar to our cluster, but the processor clock is faster (3.2 GHz), the local storage is a nonstriped SCSI disk, and the network topography is such that groups of nodes share the same gigabit ethernet connection. This results in lower efficiency, due to the less efficient network and the slower speed of local storage (a single disk).

Acknowledgment. This work was supported by the National Science Foundation under grant number CHE-0515922 and by the Mildred B. Cooper Chair at the University of Arkansas. Acquisition of the Red Diamond supercomputer was supported in part by the National Science Foundation under award number MRI-0421099.

References

(1) Bartlett, R. J.; Purvis, G. D., III *Int. J. Quantum Chem.* **1978**, *14*, 561.

(2) Pople, J. A.; Krishnan, R.; Schlegel, H. B.; Binkley, J. S *Int. J. Quantum Chem.* **1978**, *14*, 545.

(3) Taylor, P. R.; Bacskay, G. B.; Hush, N. S.; Hurley, A. C. *J. Chem. Phys.* **1978**, *69*, 1971.

(4) Sinanoğlu, O. *J. Chem. Phys.* **1962**, *36*, 706.

(5) Čížek, J. *J. Chem. Phys.* **1969**, *45*, 4256.

(6) Pople, J. A.; Head-Gordon, M.; Raghavachari, K. *J. Chem. Phys.* **1987**, *87*, 5968.

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

(8) Urban, M.; Noga, J.; Cole, S. J.; Bartlett, R. J. *J. Chem. Phys.* **1985**, *83*, 4041–4048.

(9) Pople, J. A.; Head-Gordon, M.; Raghavachari, K. *J. Chem. Phys.* **1987**, *87*, 5968.

(10) *PQS*; version 3.2, Parallel Quantum Solutions: Fayetteville, AR.

(11) Janowski, T.; Ford, A. R.; Pulay, P. *J. Chem. Theory Comput.* **2007**, *3*, 1368–1377.

(12) Meyer, W. *J. Chem. Phys.* **1973**, *58*, 1017–1035.

(13) Pulay, P.; Saebo, S. *Theor. Chim. Acta* **1985**, *69*, 357–368.

(14) Amos, R. D.; Bernhardsson, A.; Berning, A.; Celani, P.; Cooper, D. L.; Deegan, M. J. O.; Dobbyn, A. J.; Eckert, F.; Hampel, C.; Hetzer, G.; Knowles, P. J.; Korona, T.; Lindh, R.; Lloyd, A.; McNicholas, S. J.; Manby, F. R.; Meyer, W.; Mura, M. E.; Nicklass, A.; Palmieri, P.; Pitzer, R.; Rauhut, G.; Schtz M. Schumann, U.; Stoll, H.; Stone, A. J.; Tarroni, R.; Thornteinsson, T.; Werner, J.-J. *MOLPRO*, a package of ab initio programs designed by Werner H.-J. and Knowles, P. J.; version 2002.1; 2002.

(15) Kendall, R. A.; Apra, 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*, 260–283.

(16) Harding, M. E.; Metzroth, T.; Gauss, J.; Auer, A. A. *J. Chem. Theory Comput.* **2007**, *4*, 64–74.

(17) Olson, R. M.; Bentz, J. L.; Kendall, R. A.; Schmidt, M. W.; Gordon, M. S. *J. Chem. Theory Comput.* **2007**, *3*, 1312–1328.

(18) Lotrich, V.; Flocke, N.; Yau, A.; Perera, A.; Deumens, E.; Bartlett, R. J. ACES III: Parallel Implementation of Electronic Structure Energy, Gradient and Hessian Calculations. In *48th Sanibel Symposium*; St. Simons Island, GA, Feb 21–26, 2008.

(19) Kobayashi, R.; Rendell, A. P. *Chem. Phys. Lett.* **1996**, *265*, 1–11.

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

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

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

(23) Rendell, A. P.; Lee, T. J.; Komornicki, A. *Chem. Phys. Lett.* **1991**, *178*, 462–470.

(24) Goto, K.; van de Geijn, R. *ACM Trans. Math. Software*, submitted for publication.

(25) Meyer, W. *J. Chem. Phys.* **1976**, *64*, 2901.

(26) Pulay, P.; Saebo, S.; Meyer, W. *J. Chem. Phys.* **1984**, *81*, 1901.

(27) Ford, A. R.; Janowski, T.; Pulay, P. *J. Comput. Chem.* **2007**, *28*, 1215–1220.

(28) Helgaker, T.; Jørgensen, P.; Olsen, J. *Molecular Electronic-Structure Theory*, first ed.; John Wiley & Sons Ltd.: New York, 2000; Chapter 15, pp 817−883.

(29) Saebo, S.; Pulay, P. *J. Chem. Phys.* **1987**, *86*, 914–922.

(30) Hirata, S. *J. Phys. Chem.* **2003**, *107*, 9887–9897.

(31) Schütz, M.; Lindh, R.; Werner, H.-J. *Mol. Phys.* **1999**, *96*, 719–733.

(32) Janowski, T.; Pulay, P. *Chem. Phys. Lett.* **2007**, *447*, 27–32.

(33) Schütz, M.; Werner, H.-J. *J. Chem. Phys.* **2001**, *114*, 661–681.

(34) Baker, J.; Pulay, P. *J. Comput. Chem.* **2002**, *23*, 1150–1156.

(35) Pulay, P.; Saebo, S.; Wolinski, K. *Chem. Phys. Lett.* **2001**, *344*, 543–552.

(36) Lee, T. J.; Rendell, A. P.; Taylor, P. R. *J. Phys. Chem.* **1989**, *94*, 5463–5468.

(37) Schütz, M.; Werner, H.-J. *Chem. Phys. Lett.* **2000**, *318*, 370–378.

(38) Schäfer, A.; Horn, H.; Ahlrichs, R. *J. Chem. Phys.* **1992**, *97*, 2571.

(39) Weigend, F.; Ahlrichs, R. *Phys. Chem. Chem. Phys.* **2005**, *7*, 3297.

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

CT800142F