JCTC Journal of Chemical Theory and Computation

Article

pubs.acs.org/JCTC

Efficient Sparse Matrix Algorithm to Speed Up the Calculation of the Ladder Term in Coupled Cluster Programs

Zoltán Pillió, Attila Tajti, and Péter G. Szalay*

Laboratory of Theoretical Chemistry, Institute of Chemistry, Eötvös University, P.O. Box 32, H-1518, Budapest 112, Hungary

ABSTRACT: A new algorithm is presented for the calculation of the ladder-type term of the coupled cluster singles and doubles (CCSD) equations using two-electron integrals in atomic orbital (AO) basis. The method is based on an orbital grouping scheme, which results in an optimal partitioning of the AO integral matrix into sparse and dense blocks allowing efficient matrix multiplication. Carefully chosen numerical tests have been performed to analyze the performance of all aspects of the new algorithm. It is shown that the suggested scheme allows an efficient utilization of modern highly parallel architectures and devices in CCSD calculations. Details of the implementation in the development version of CFOUR quantum chemical program package are also presented.

INTRODUCTION

Coupled cluster (CC) theory[1] provides a very accurate solution to the electronic structure problem.[2−6] The hierarchical approximations in CC theory make it capable of obtaining high level results: starting with the singles and doubles approximation (CCSD),[7] one can successively include higher excitations and improve the description accordingly.[2,6] The method, including approximate triple excitations CCSD(T),[8,9] is often referred to as the "golden standard" of quantum chemistry.

Many program packages offer efficient implementation of CCSD and CCSD(T) such as ACES II,[10] CFOUR,[11] MOLPRO,[12] MolCAS,[13] PQS,[14] Psi3,[15] MRCC,[16,17] NWChem,[18] ACES III[19] etc., and they use different algorithms to obtain a nearly optimal execution. Most of the computational tasks of the CCSD calculations[7] can be cast into matrix−matrix multiplications. The most time-consuming multiplications are the ladder-type contractions

$$Z_{ab}^{ij} = \sum_{cd} T_{cd}^{ij} W_{ab}^{cd} \tag{1}$$

where W is the supermatrix of the two-electron integrals in the MO representation (henceforth integral matrix), T is the matrix of the double excitation amplitudes, and Z is the matrix of the intermediates (i,j denotes occupied, and a, b, c, d denote virtual spin orbitals). Analogously, this multiplication can also be formulated with AO integrals,[20] avoiding the expensive AO to MO transformation of the all-virtual integrals (W_{ab}^{cd}). In this case, the a, b, c, d virtual spin orbital indices are replaced by the σ, ρ, μ, ν AO indices in eq 1; that is, the amplitude and intermediate matrices are half-transformed quantities, while the integral matrix W is fully in AO basis. This is advantageous because the AO integral matrix is more sparse than the MO one.[20]

Unfortunately, not all implementations are well documented in the literature. As already cited, the AO basis implementation has been described by Hampel and Werner.[20] Integral direct version has been reported by Schütz, Lindh, and Werner.[21] It was Kobayashi and Rendell[22] who described a massively parallel implementation for the first time. Recently, Janowski and Pulay[23,24] and Harding, Metzroth, and Gauss[25] described new parallel implementations in detail. In the latter, the parallelization is done over AO basis functions, with the different threads working on batches of basis functions. Janowski and Pulay[23,24] investigated different aspects of such algorithms including the sparsity of the integral matrix (for details see later). In the same year, Lotrich et al.[26] presented a massively parallel implementation of several Many-Body methods including CCSD and CCSD(T).

For the sake of completeness, we mention that Cholesky decomposition of the integral matrix is used in MolCAS to reduce the cost of the ladder (and other) contractions.[13]

In this paper, we reconsider the implementation of the CCSD method by concentrating mainly on execution on modern computer architectures and trying to reduce operation count and support parallel execution at the same time. Two important ingredients of our new algorithm are an efficient blocking scheme to obtain optimal sparse-dense structure of the integral matrix and the independent multiplication of these two parts.

OVERVIEW OF THE PROBLEM

Structure of the Integral Matrix. In eq 1, the integral matrix is constructed from two-electron integrals as

$$W(\mathcal{I}(\sigma,\rho),\ \mathcal{I}(\mu,\nu)) = \langle\sigma(1)\rho(2)|\mu(1)\nu(2)\rangle \equiv \langle\sigma\rho|\mu\nu\rangle \tag{2}$$

with $\mathcal{I}$ being the indexing function, generally:

$$\mathcal{I}(\sigma,\rho) = \sigma N + \rho \tag{3}$$

(N is the number of basis functions).

A simple matrix−matrix multiplication of this integral matrix in the AO basis and the amplitudes is not the best strategy for two reasons: (i) The matrix, in particular, in the case of larger molecules, is rather sparse; Therefore, a lot of elements need

Received: May 16, 2012
Published: August 8, 2012

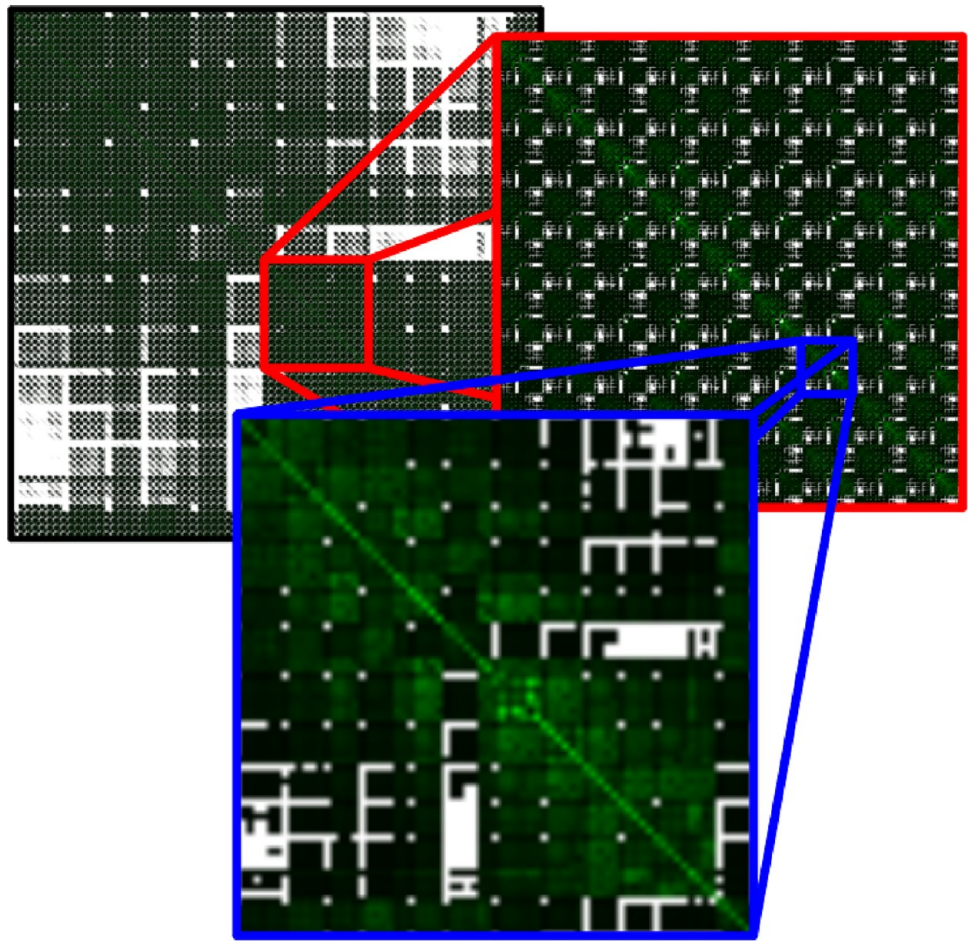

Figure 1. Absolute values of the elements of the full two-electron integral supermatrix (black border), a part of this matrix (red border), and a submatrix with the same σ and μ indices (blue border). Neglected integrals (actually less than 10^{-14}) are marked with white, while for other integrals a logarithmic color code is used from black (10^{-14}) to bright green (greater than 10^{0}). The matrix and the submatrix share the same pattern. (Arginine with STO-3G basis.)

not to be considered. (ii) After their evaluation, the integrals are not stored in a matrix form but rather (at least from the matrix multiplications point of view) randomly, and a sorting would be too expensive.

To get an idea about the structure of the integral matrix, we suggest the following trivial procedure. First, one can approximate the size of the integrals by the Cauchy-Schwarz formula,[27] which provides an upper bound for the value of the integrals:

$$|\langle\sigma(1)\rho(2)|\mu(1)\nu(2)\rangle| \leq \sqrt{\langle\sigma\sigma|\mu\mu\rangle\langle\rho\rho|\nu\nu\rangle} \tag{4}$$

That is, they can be estimated using two special integrals.

Second, we introduce the term submatrix: a submatrix includes integrals having the same σ and μ indices. This means that the whole integral matrix has $N \times N$ submatrices, each of the size $N \times N$. Based on eq 4, if the integral $\langle\sigma\sigma|\mu\mu\rangle$ is small; that is, σ and μ are far in space (at this point, it is not necessary to define the term "far", since in the proposed algorithm no integral will be neglected), a submatrix will have only vanishing (or very small) elements. On the other hand, for nonvanishing submatrices (corresponding to σ and μ indices close in space, i.e. $\langle\sigma\sigma|\mu\mu\rangle$ being nonvanishing), the $\langle\sigma\rho|\mu\nu\rangle$ integrals will be negligible only if $\langle\rho\rho|\nu\nu\rangle$ is small, that is, if ρ and ν are far enough from each other. This means that submatrices show similar pattern of vanishing elements as the whole matrix does. This is represented in Figure 1. We will use this rather trivial property of the integral matrix later.

The figure also shows clearly that the integral matrix is very sparse, and the nonzero elements follow an apparent structure. We can recognize submatrices with a large number of nonzero elements. Clearly, for the submatrices with a large number of zero elements, efficient handling is required, since both the storage of, and the multiplication by, the zero elements are unnecessary.

Sparse Matrix Handling. Using various indexing schemes, it is possible to store only the nonzero elements of sparse matrices. The simplest (though not necessarily the most efficient) method is to store the values and the corresponding indices. This is a common choice in many quantum chemical programs, such as CFOUR[11] and its predecessor ACES II.[10] In this case, the sparse matrix−matrix multiplication can be formulated according to Scheme 1, where loop C runs over the occupied−occupied composite index.

The inner loop is a scalar-vector multiplication (DAXPY[28]), which can be parallelized easily. On the other hand, the outer loop uses two indices, which are unrelated in the consecutive steps. Consequently, the necessary Z and T_2 elements can be reached by disordered memory access only, so efficient caching is impossible.

dx.doi.org/10.1021/ct300391t | J. Chem. Theory Comput. 2012, 8, 3108−3118

Scheme 1

```
LOOP nonzero integrals (value, index1, index2)
 LOOP C
  Z(C,index1) = Z(C,index1) + value * T2(C,index2)
 END LOOP
END LOOP
```

Further complication arises from the permutational symmetry of the two-electron integrals. Usually, only the unique AO integrals are stored, and an integral needs to be used in several (up to 8) such loops corresponding to different T and Z elements.

Disregarding the permutational symmetry, this algorithm has the optimal operation count; that is, the integrals are multiplied only as many times as necessary. Moreover, this scheme can easily be parallelized by assigning batches of integrals to the different processors.[25] On the other hand, efficient implementation on a massively parallel device, such as Graphics Processing Unit (GPU), Field-Programmable Gate Array (FPGA), or many-core CPUs[29] with vector instructions such as SSE or AVX, is not possible, because scalar−vector multiplications need too many memory operations.

The problem we face in coupled cluster calculations is related to other procedures used in other fields of quantum chemistry. For example, in linear scaling SCF algorithms, both the Fock and density matrices are sparse, and efficient algorithms use blocking schemes; that is, a full matrix multiplication is split into multiplication of small(er) blocks, which helps to identify sparse and dense blocks and has further advantages (e.g., in memory requirement). To define the blocks, the basis functions are typically assigned into groups, and the integral matrix is blocked according to these orbital groups.[30,31] The grouping of the basis functions can be done in different ways. One way is to assign all functions centered on the same atom into a group.[30] The problem with this grouping is that the resulting integral blocks are eventually too small and of different size, which hampers the efficient execution of BLAS GEMM[32] subroutines. Assigning the basis functions of more than one atom to a group, for example by boxing technique,[31] can solve the size problem, but the size of the groups and therefore the dimension of integral blocks still varies with the atom types. In the perturbation theory and coupled-cluster implementation, ACES III also uses blocks, and it is claimed that this blocking results in an efficient parallelization over tens of thousands of processors.[19,26]

Janowski and Pulay[23] recognized that, even in case of the partially sparse AO integral matrix, full matrix multiplication can be advantageous since the computer hardware is much more efficient in this case, compensating for the unnecessary manipulation of zero elements. To use the hardware efficiently, but also make use of sparsity, they decompose the whole integral matrix into blocks and replace the full multiplication with that of the blocks.[23] One of the advantages of this scheme is that the very sparse blocks can be processed element by element, while in case of the dense blocks the matrix−matrix multiplication can be used efficiently.[23] They also suggest a reordering of the basis set according to the topology of the molecule (essentially moving the basis functions centered on the same atom next to each other) to get better dense/sparse separation.

Methods that aim to restructure matrices in order to get a better ratio of sparse and dense blocks by reordering the indices are often called as sparse matrix partitioning.[33] Using symmetry in quantum chemical calculations is a trivial example for sparse matrix partitioning (for coupled cluster implementation, see ref 34). Reordering of the basis functions as suggested by Janowski and Pulay[23] can also be considered as a matrix partitioning technique. However, beside the topology of the atoms and the corresponding orbitals, other factors, such as orbital exponents, orbital types (s, p, d, ...), contraction scheme, etc., determine the actual value of the integral and therefore influence the efficiency of such a procedures. We therefore suggest an automated procedure to find the best ordering of the basis functions, resulting in the optimal separation of dense and sparse blocks of the integral matrix.

Sparse Matrix Multiplication with Optimal Partitioning. General sparse matrix partitioning techniques are based on identifying the nearly related rows and/or columns of the sparse matrix and reordering them accordingly. This, however, requires that the whole integral matrix is kept in the memory, which is clearly an impossible task for large molecules.

The first ingredient of our algorithm suggests a solution for this storage problem. As discussed above, the integral matrix has a special structure with submatrices having similar structure concerning sparsity than the whole one (see Figure 1 and discussion thereof). This means that the matrix partitioning can be performed on the submatrices, which are substantially smaller and can easily be stored in the memory. Optimal ordering of the basis functions will result automatically in optimal partitioning of the entire integral matrix. To provide a better model for the whole matrix, the $\langle\sigma\sigma|\mu\mu\rangle$ integrals are used as elements of the submatrix. This corresponds to an approximation of the two-electron integrals according to the Cauchy−Schwartz formula (see eq 4).

We used three different grouping methods based on the submatrices.

(a). Block Diagonal Grouping. In this scheme, those basis functions are assigned to a group that all have "large" integral values with each other; that is, the functions are "close". The "closeness" of all basis functions in a group can be measured by the quantity $\rho_\alpha = \Sigma_{\sigma\neq\mu} \langle\sigma\sigma|\mu\mu\rangle$, where σ and μ belong to a group α. Therefore, the criteria used for reordering the matrix is that the sum of the ρ_α values over all groups is maximized. Because the considered elements in the blocks are situated only along the diagonal of the submatrix, we refer to this scheme as "block diagonal grouping".

(b). Total Block Grouping. In this algorithm, instead of using the values of the integrals $\langle\sigma\sigma|\mu\mu\rangle$ as elements of the submatrix, we use a binary representation: elements of this binary matrix will be 1 unless the corresponding integral has been deleted in the usual prescreening of the AO integrals. This means that we replace the integral matrix with a simplified one, where integrals are either zero or nonzero (for simplicity, a value of 1 is used in this case); that is, we do not distinguish the nonzero elements according to their value. This is, however, a legitim simplification, since we need to perform multiplication of all nonzero integrals irrespective of their magnitude. Note that the structure of this matrix and the result of the grouping will, of course, depend somewhat on the value of the threshold used for the prescreening.

In the search for the optimal matrix, the goal is to collect the nonzero elements of the binary matrix into the fewest possible

blocks. We count the number of nonzero values within each block and maximize the square of their sum over all blocks.

(c). METIS.[35] We have also tested some standard library procedures to obtain optimal partitioning with which to compare the described simpler schemes. In particular, the package called METIS[35] was used. METIS is a standard collection of graph and sparse matrix partitioner algorithms (note that sparse matrices can be represented by graphs, with edges representing rows and columns and vertices representing the nonzero matrix elements). We have tested the gpmetis and the ndmetis programs of METIS[35] with different optional parameters. The ndmetis program results in a so-called fill-reduced form of the matrix, which has the nonzero elements close to each other. For our purpose, this procedure has the disadvantage that quite large nonzero blocks can arise, which need to be split up into smaller blocks optimal for the matrix multiplication (see later). The gpmetis program partitions the matrix into optimal block structure, but the size of the groups defining the blocks also varies. In this case, it is possible to control the size of groups to some extent, but the resulting blocks are not always equal in size. Therefore, here too, we have to change the size of the blocks in an additional step (see later). In the case of gpmetis, the best result has been obtained using a recursive "bisectioning" scheme (-ptype=rb) and "minimize the edgecut" partitioning objective (-objtype=cut). In the following, we will use the results of this parametrization when referring to gpmetis.

(d). Hand Sorting. To test the spatial ordering schemes such as the boxing technique[31] discussed, we also performed a partitioning by simple ordering of the basis functions prior to the calculations. We could exploit the fact that in CFOUR the initial order of the basis functions corresponds to the sequence of atoms in the input file. By ordering close-lying atoms after each other, spatial ordering could be achieved (note the close relationship of this procedure to the one by Janowsky and Pulay[23]). In this "trivial" procedure, the original orbital sequences are cut into equal parts to obtain groups of orbitals defining the integral blocks.

IMPLEMENTATION

In this section, we present some details of our implementation of the CC ladder term in the development version of CFOUR.[11] We will discuss the necessary steps such as grouping the orbitals, sorting the integrals accordingly, handling of the permutational symmetry, and finally performing the multiplication.

Orbital Grouping. As described, we have tested four different grouping methods. As we mentioned previously, in CFOUR the initial order of orbitals corresponds to the ordering of atoms in the input. Since, very often, the initial order of atoms resembles their spatial position, we used random distribution as a starting point for grouping algorithms, except in the case of hand sorting, of course. This randomization was necessary to obtain an unbiased reference point for the different algorithms. If we do not do that, the initial grouping could be optimal (or closely optimal), and we will not see the performance of the difference grouping schemes. The "hand sorting" algorithm is included in the study to see how efficient grouping can be obtained by simple (but not necessarily trivial) manipulation of the input.

Block diagonal grouping and total block grouping algorithms loop over all pairs of orbital groups, and for each pair, it checks all possible swaps of orbitals between the two groups. A swap is accepted if the target function is improved. If any swap is accepted in an iteration, the search starts again. Convergence is ensured by the fact that the target functions of both schemes are bounded from above. Note also that this algorithm scales as N^2 (block diagonal grouping) and N^3 (total block grouping) where N is the number of basis functions. The typical iteration count is four to six. The iteration count of METIS with recursive bisectioning scheme is proportional to $\log_2 N$ (in the test calculations presented below, the iteration count was 51 and 111). The number of elementary steps is proportional to the connectivity of the two parts, which is proportional to N for real sparse matrices. The measured computational time was negligible compared to other parts of the CC calculation for all of the algorithms.

Integral Sorting. Having the optimal grouping of the orbitals, the integral matrix needs to be reordered. Originally, an integral is represented by four orbital labels. Now, an integral will be represented by four group labels and four subgroup labels (i.e., the serial number of an orbital within a group). The integrals with the same group labels compose an integral block. The integral blocks are indexed by these group labels. It is straightforward to characterize the blocks as "dense" and "sparse" depending on the number of nonzero elements and the two types can be processed by different algorithms.

The integral sorting is a critical step of any non-integral-direct correlated calculations (such as the one in CFOUR) since it can be I/O bound for large systems. The number of AO integrals is very large; typically, the integral values are stored as a list of the integral values together with the corresponding indices. This is advantageous because only nonzero elements need to be stored and also redundant integrals (those related by permutation of indices) can be skipped. On the other hand, the integrals are practically randomly given in these lists. We use a three-step bucket sort to get the integral blocks. The first and the second passes go according to the first two labels. Although the use of three steps increase the overall number of I/O operations, we have found that—assuming an orbital group size of eight (see later)—unifying the first two passes would result in a too large number of buckets, rendering the input from the direct access file too slow, which causes a worse overall I/O performance.

In the last step of the sorting, all nonzero elements of the processed block are available in the memory. If it contains enough nonzero items, it is stored as an array including the zeros (continuous part); otherwise, it will be stored as a list of the nonzero values with the corresponding labels (sparse part).

The ladder-term multiplication in the AO basis also requires the half-transformation of the amplitudes (T) from, and the back transformation of intermediate quantity (Z) to the MO basis.[20] Because in our algorithm the AO orbital labels are reordered to define an optimal block structure of W, the half-transformation also needs to consider the reordering, which can simply be obtained by reordering the rows of the MO-AO coefficient matrix.

Permutational Symmetry. Because originally only the nonredundant integrals are stored, the permutational symmetry needs also to be considered during the sort. The permuted matrix elements can be generated by label swap, which decomposes into a group label and a subgroup label swap. There are two cases to consider here. If the group labels corresponding to the swapped labels differ, the permuted matrix element will belong to a different block. We refer to this as "interblock" redundancy. In this case, a new block can be

generated by permuting the elements of a given block. These redundant blocks can be used in the multiplication immediately after their generation; therefore, these do not need to be stored. In the other case, that is, when the swapped group indices are the same, the permuted matrix element will belong to the same block. We refer to this as "intrablock" redundancy. These redundant elements can immediately be copied to the appropriate position of the block before it is written to disk. Storing the intrablock redundant elements does not impose any bottleneck in the storage cost, since the number of redundant elements processed as interblock redundant is much larger than that of the intrablock ones for larger calculations. The scheme of the handling of redundancy is demonstrated in Figure 2.

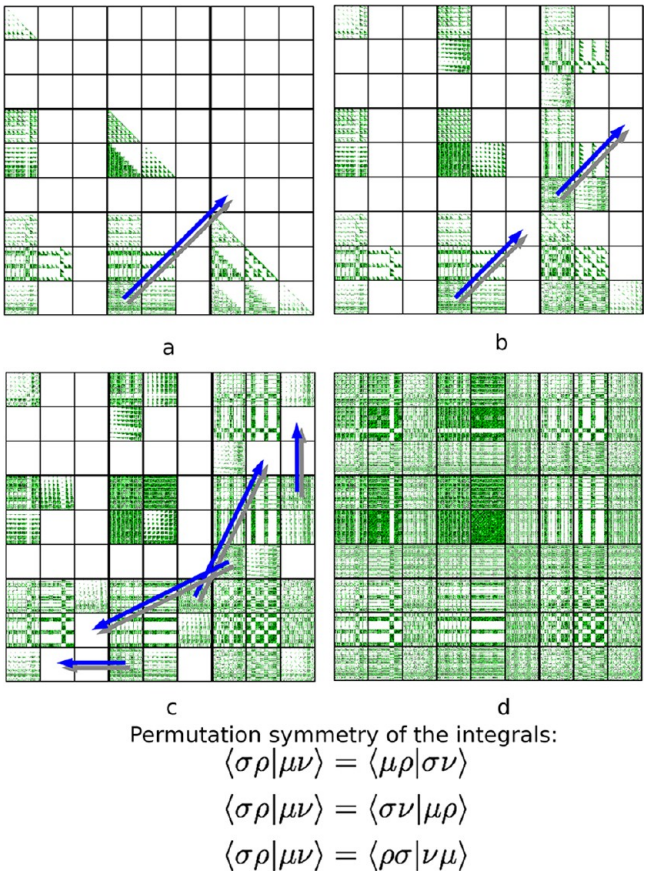

Figure 2. Three permutation steps used in our algorithm to generate redundant representation (d) from a stored unique block (a). The three steps correspond to the three types of permutational symmetry of two-electron integrals listed on the bottom of the figure. The third step is only necessary in UHF case. (H_2O with cc-pVDZ basis.)

Note that, using blocks, the handling of the redundancy is more efficient: without the block structure, the permuted matrix elements are scattered around the whole integral matrix, which might cause high memory latency. On the other hand, in case of reasonable (not too large) block size, the operation is cacheable, and therefore, this step will be much more efficient. As it is shown in Figure 3, the algorithm that builds the integral matrix was significantly faster using block sizes of 16, 64, or 256 than using a block size of 1024 or the original indexing.

Ladder Term Calculation. The matrix multiplication of the ladder term (eq 1) can be decomposed into multiplications of smaller blocks.[23] As discussed, the integral blocks can be classified as dense (having predominantly nonzero elements), and as sparse (having only a few nonzero elements). Our algorithm handles these blocks separately. For the dense blocks, there are two steps: after reading a block, the redundant block(s) (interblock redundancy) are also formed, and with these, the multiplications are also performed. Note that since T has been reordered according to the new AO indices, the multiplication only requires a particular segment of the T quantity.

In the case of sparse blocks, where only the nonzero elements are stored, the multiplication is performed as in the original algorithm (see Scheme 1).

TEST CALCULATIONS

Efficiency of the Grouping. Figure 4 and Table 1 present data on the performance of the different grouping algoritms. In these test calculations, the 1,3,5,7,9,11-dodecahexaene ($C_{12}H_{14}$) molecule in the all-trans configurations was used, which is large enough to show notable sparsity of the integral matrix. Two basis sets were used: besides the cc-pVDZ,[36] we also applied aug-cc-pVDZ,[37] which includes diffuse functions, as well. Figure 4 shows the distribution of the blocks with respect to the number of nonzero elements they contain: the more elements the dense blocks contain, and/or the less elements the sparse blocks contain, the more efficient is the algorithm in separating sparse and dense blocks. With respect to efficiency, the middle of the graph also is important because the corresponding blocks include about the same number of zero and nonzero elements. Storing the latter causes extra work not only in storage but also in multiplication. Table 1 compares the execution time of the ladder type contraction and the ratio of stored zero and nonzero integrals. The faster the calculation runs, the better the sparse and dense blocks are separated. Similarly, the fewer zero elements are stored, the more efficiently are the zero elements transferred into sparse blocks.

The same conclusion can be drawn from these data: all four grouping techniques are capable to separate the very dense and very sparse blocks. In all cases, there is only a small number of blocks in the middle of Figure 4. Still, the "total blocking" algorithm seems to be the most efficient, showing the largest density for very dense and very sparse matrices (4), the shortest execution time and lowest rate of the stored zero integrals (Table 1). METIS and "hand sorting" also work very well, while "block diagonal" sorting falls somewhat behind.

The above conclusion is valid for both valence and diffuse basis sets, and it seems that "total block grouping" outperforms the other techniques even more if diffuse functions are included in the basis set. The difficulty caused by diffuse functions in the grouping was noticed by Janowski and Pulay:[23] having both ordinary and diffuse functions in the same group causes efficiency problems. They solved this problem by putting the diffuse functions into different groups. This "hand-made" separation is not necessary in our algorithm because the target functions of the blocking procedures can treat any type of function on the same footing, and diffuse functions will be placed into those groups where the most efficient matrix structure is obtained.

According to these results, we suggest the "total block grouping" as the default choice in our implementation.

Efficiency of the General Matrix Multiplication (GEMM).[32] The efficiency of matrix multiplication depends on the size of the matrices. Since in our algorithm the integral matrix is partitioned into blocks, the matrix multiplication is characterized by the block size, but also by the length of the

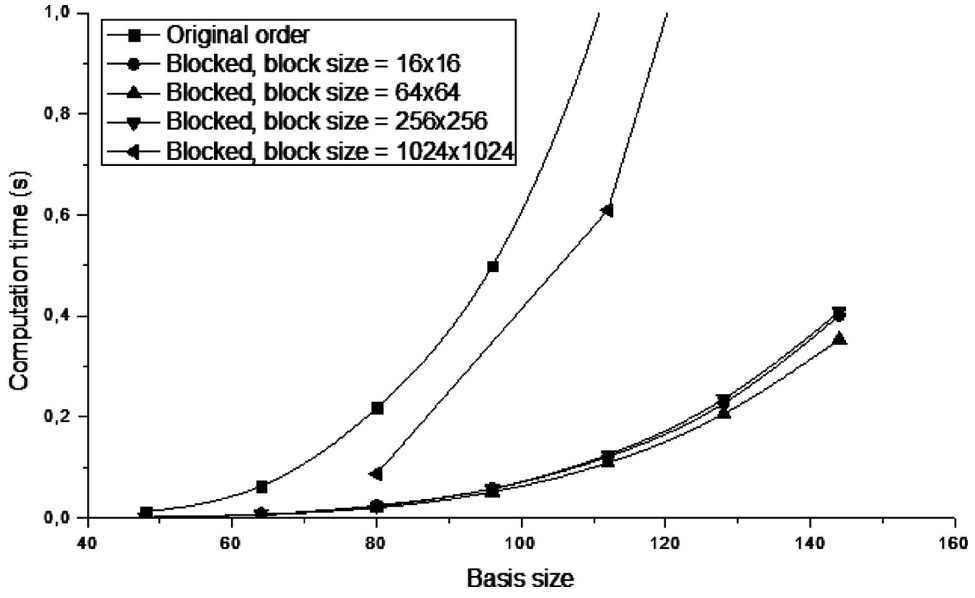

Figure 3. Execution time of the algorithm, which builds the integral matrix from the nonredundant matrix elements, using different matrix size and storage techniques (original indexing and block structure with different block sizes).

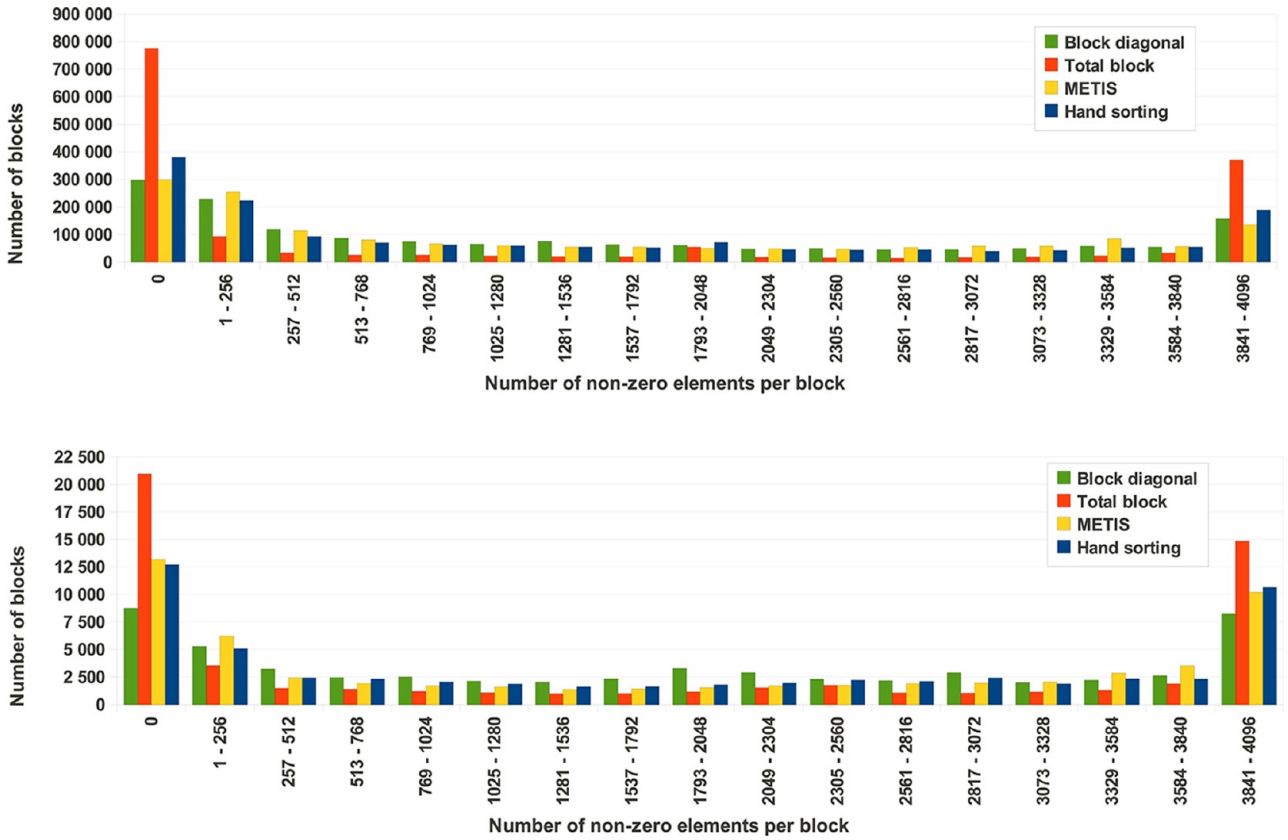

Figure 4. Distribution of integral block sparsity for the $C_{12}H_{14}$ test molecule using cc-pVDZ (top) and aug-cc-pVDZ (bottom) basis sets.

occupied−occupied composite index of the T_2 amplitudes (see eq 1).

To find the optimal block size and to decide whether the T_2 amplitudes need to be partitioned into blocks as well, we have performed a series of test calculations using different sizes of the blocks. The integral block was assumed to be a square matrix, its dimension corresponds to the square of the integral group size (k^2), while the outer index of the T_2 amplitude matrix (M) was also varied. Note that this index corresponds to the square of the number of occupied orbitals (if no blocking of T is performed).

By various dimensions of the matrix multiplications, we have tested how the commonly available parallelization strategies perform: MPI (Message Passing Interface)[38] and OpenMP[39] models were used. In short, the former is a distributed memory scheme, which in our case means that the different processors

Table 1. Comparison of Different Grouping Methods

grouping method	block diagonal	total block	METIS	hand sorting
	Timing			
$C_{12}H_{14}$, cc-pVDZ	80.0 s	68.5 s	72.5 s	75.5 s
$C_{12}H_{14}$, aug-cc-pVDZ	1988 s	1270 s	1911 s	1872 s
	Ratio of Stored Zero and Nonzero Integrals			
$C_{12}H_{14}$, cc-pVDZ	0.51	0.19	0.34	0.40
$C_{12}H_{14}$, aug-cc-pVDZ	0.66	0.21	0.63	0.57

are working on the multiplication of different blocks at the same time, while in case of OpenMP model parts of the same matrix multiplication are distributed among different processors. Our test was a simulation of the real calculation, where the matrix multiplications are executed practically randomly over the blocks (sparse blocks are skipped, while permuted ones are generated in-place). One or some multiplications with the same data would not give representative result since caching would not correspond to the real situation. Therefore, several blocks were used. Since all blocks were held in the memory, I/O did not bias the calculations. For simplicity, only one node with two processors (four cores each) was used in all calculations.

Table 2 shows the speed-up obtained with the different combination of OpenMP and MPI threads on blocks of different size with respect to a calculation performed on a single core. One can observe that in most cases speed-up increases with both block dimensions k^2 and M, but even $k^2 = 1024$ is too small to give enough work for eight processors. Efficiency of MPI with eight cores is drastically reduced compared to two or four cores, presumably because of memory bandwidth. On the other hand, OpenMP parallelization cannot be efficient with small blocks since there is not enough work to be distributed among eight or even four cores. Therefore, a mixed algorithm could be advantageous: the last two columns of the table show that both mixed strategies can give some additional speed-up—also in case of smaller matrices—with respect to both pure MPI and OpenMP calculations using the same number of cores. Note also that for the largest matrices (bottom of the table) all models give similar speed-up, quite close to the theoretical maximum.

Therefore, one can conclude that, in the investigated range, the best performance can be obtained with the largest dimensions of both k^2 and M. Concerning the latter, it does not seem to be useful to divide M into smaller pieces. Note that $M = 4096$ corresponds to 64 occupied orbitals, which is clearly around the expected maximum we can handle in real calculations. On the other hand, increasing the dimension of the integral blocks, although it would increase the speed of the multiplication, can make the number of truly dense blocks smaller, leading to a less efficient execution (see later).

Finally, the MPI parallelization on one node, even within the mixed strategy, can be much less efficient in real calculations, where the integral blocks need to be read from disk resulting possibly in an I/O bottleneck. On the other hand, combining the multinode MPI parallelization available in CFOUR[25] with the OpenMP parallelization within a node, can be a real solution. This will be further investigated below.

Optimal Identification of Dense and Sparse Blocks. There is one more parameter to be specified in our new algorithm, namely the criterion to distinguish sparse and dense blocks. Clearly, for a small number of nonzero elements, the multiplication of only the nonzero elements should be preferred, while with growing number of nonzero elements the multiplication of zeros will be compensated by the efficiency of the multiplication of the whole matrices. Figure 5 was created to obtain the best definition of sparse and dense blocks. The figure reports several quantities as a function of different criterion for sparsity ($C_{12}H_{14}$, cc-pVDZ basis): Figure 5a shows the total execution time of the ladder type multiplications, that is, the sum of the time spent with the sparse (Figure 5b), and full (Figure 5c) multiplications; Figure 5d shows the number of double-precision operations, Figure 5e demonstrates the overall performance, and finally, Figure 5f shows the number of L2 cache misses. The tests were performed for three different cases: random integral matrix with block size 8 and partitioned integral matrix using block sizes 8 and 16. In all cases, the completely zero blocks have not been considered in the calculations; the number of these blocks depends on the block size, as well as on the partitioning. Therefore, one observes some differences for the overall work

Table 2. General Matrix Multiplication Performance Using Different Matrix Sizes

	OMP[a]	1	1	1	1	2	4	8	2	4
	MPI	1	2	4	8	1	1	1	4	2
k^2	M	GFLOP/s[b]	su[c]	su[c]	su[c]	su[c]	su[c]	su[c]	su[c]	su[c]
64	256	7.65	1.76	3.37	4.65	1.61	2.42	3.42	4.29	3.19
256	256	8.08	1.89	3.31	4.50	1.78	2.93	2.96	4.12	5.12
1024	256	7.66	1.83	3.24	3.53	1.95	3.28	4.46	4.96	5.31
64	1024	7.42	1.85	3.40	5.44	1.91	2.73	4.04	5.30	4.72
256	1024	8.29	1.97	3.66	6.62	1.96	3.22	4.72	6.84	5.88
1024	1024	8.19	1.92	3.44	5.93	1.98	3.79	6.16	6.66	6.83
64	2304	7.41	1.85	3.35	5.26	1.91	2.97	4.85	5.17	5.05
256	2304	8.35	1.98	3.67	6.75	1.96	3.52	6.45	6.68	6.66
1024	2304	8.21	1.92	3.46	5.98	1.97	3.79	6.24	6.54	6.82
64	4096	7.35	1.86	3.36	5.26	1.92	3.10	5.06	5.19	5.12
256	4096	8.32	1.98	3.68	6.74	1.97	3.63	6.65	6.81	6.87
1024	4096	8.22	1.92	3.42	5.93	1.99	3.82	6.45	6.65	6.91
8192	8192	8.65	1.95	3.96	7.58	1.98	3.87	7.43	7.55	7.52

[a]Using Intel MKL 11.1 inbuild parallelization. [b]Computation performance of one processor core. Theoretical performance of Intel Xeon E5420 processor, which was used in our test calculations, is 10 GFLOP/s/core, calculated using the formula: GFLOP/s/core =2.5 GHz (frequency) × 4 double-precision FLOP/cycle. [c]Speed-up.

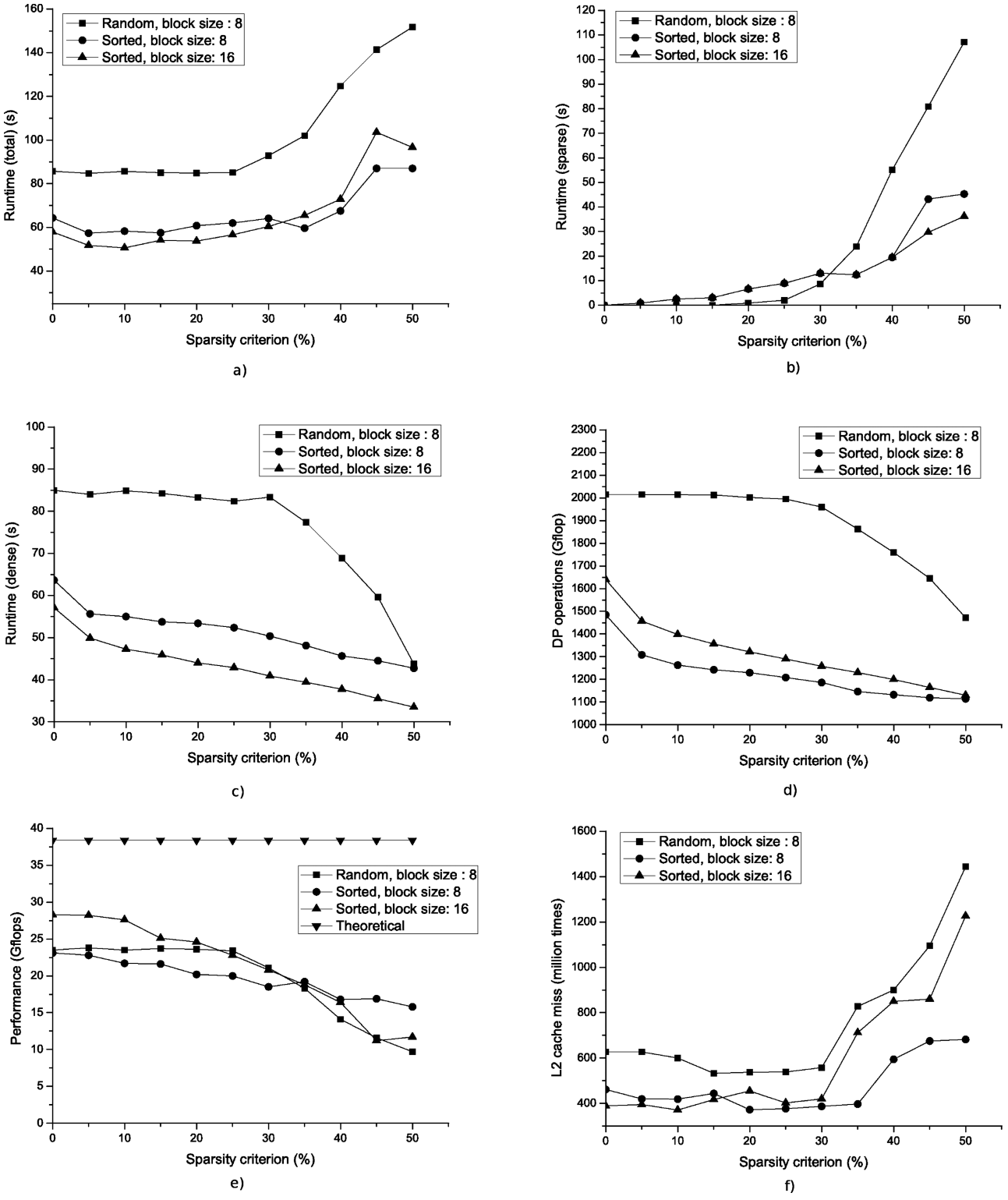

Figure 5. Effect of sparsity criterion on CC ladder term execution times. ($C_{12}H_{14}$ with cc-pVDZ basis.)

performed in the different tests. (Note that inclusion of zero blocks would introduce an even larger bias in these tests.)

As expected, the time spent with the sparse multiplication (Figure 5b) grows, while that for the dense multiplication (Figure 5c) decreases with increasing sparsity criterion. As a result, there is a minimum of the total execution time (Figure 5a). The behavior of the two test cases corresponding to block sizes 8 and 16 are very similar, while the minimum is less pronounced in case of random distribution of the integrals. This is because in the latter case there are only a few blocks with small number of nonzero integrals and, therefore, applying the sparsity criterion does not help in reducing the number of operations (Figure 5d). On the other hand, the results show how effectively the partitioning works by allowing an efficient use of the sparse multiplication. Still, it is somewhat surprising that the use of sparse multiplication for the sparsest blocks

dx.doi.org/10.1021/ct300391t | J. Chem. Theory Comput. 2012, 8, 3108−3118

Table 3. Test Calculations for Conjugated Polyenes Using Different Basis, Orbital Group Size and Number of Processor Cores (Execution Times in Seconds)

	original	original	original, MPI[b]	trivial OMP	new	new	new	new	new	new	new	new
orbital group size					4	4	8	8	16	16	32	32
no. of cores[a]	1	1(4)[c]	4	4	1	4	1	4	1	4	1	4
					Integral Sort							
C_8H_{10}, cc-pVDZ					40.7	42.6	50.6	50.8	72.5	61.3	65.9	60.8
$C_{12}H_{14}$, cc-pVDZ					147.8	129.4	120.4	120.7	124.6	144.7	115.7	126.2
$C_{16}H_{18}$, cc-pVDZ					228.7	226.8	211.0	230.9	270.5	254.7	269.1	267.4
$C_{20}H_{22}$, cc-pVDZ					382.6	415.4	401.5	382.4	396.3	393.4	-	-
C_8H_{10}, aug-cc-pVDZ					292.6	284.9	318.9	283.1	295.4	286.6	352.4	381.9
$C_{12}H_{14}$, aug-cc-pVDZ					972.6	1000.1	970.3	948.5	1830.6	1863.7	1729.6	1617.2
C_8H_{10}, cc-pVTZ					681.1	696.3	691.0	697.4	874.4	869.4	1234.1	1364.0
$C_{12}H_{14}$, cc-pVTZ					2010.6	2063.4	2101.4	1936.5	3328.8	3399.9	3895.8	3527.4
					Ladder Type							
C_8H_{10}, cc-pVDZ	77.4	75.3	35.8	61.4	35.1	24.9	24.5	9.6	32.6	12.8	42.4	17.0
$C_{12}H_{14}$, cc-pVDZ	557.0	550.3	266.9	254.4	224.9	134.4	158.9	55.3	182.3	60.8	276.5	107.4
$C_{16}H_{18}$, cc-pVDZ	2054.0	2068.5	3746.5	905.6	822.3	452.7	638.1	230.4	715.3	279.0	1164.3	533.3
$C_{20}H_{22}$, cc-pVDZ	5658.1	5405.8		2474.5	2170.4	1190.5	1587.5	564.8	1653.6	594.0		
C_8H_{10}, aug-cc-pVDZ	658.3	671.6	243.5	403.9	307.0	204.0	211.3	83.0	225.8	84.8	342.5	168.6
$C_{12}H_{14}$, aug-cc-pVDZ	5414.1	5443.2	4983.1	2530.0	2194.6	1322.4	1563.2	572.5	1591.1	556.6	1929.5	774.3
C_8H_{10}, cc-pVTZ	1622.3	1647.5	681.6	940.4	745.5	513.8	529.6	210.5	590.1	235.1	793.6	411.6
$C_{12}H_{14}$, cc-pVTZ	10372.3	10486.3		4597.9	4191.6	2615.3	3211.5	1180.0	3286.3	1087.9	4552.8	1884.8
					CC Loop							
C_8H_{10}, cc-pVDZ	107.1	90.8	84.9	78.3	64.6	40.5	56.7	27.4	65.3	31.1	263.6	36.4
$C_{12}H_{14}$, cc-pVDZ	818.8	676.3	1256.3	384.4	480.4	266.1	436.3	193.5	443.2	213.0	541.7	244.9
$C_{16}H_{18}$, cc-pVDZ	3409.8	2655.9	15012.4	1578.4	2221.9	1105.9	2069.5	921.2	2149.4	954.7	2593.0	1210.8
$C_{20}H_{22}$, cc-pVDZ	10637.8	7495.1		4661.3	7041.0	3415.8	6397.5	2798.1	6479.9	2919.2		
C_8H_{10}, aug-cc-pVDZ	818.9	748.9	1079.1	487.6	473.5	295.0	386.6	171.4	401.9	181.6	511.1	264.6
$C_{12}H_{14}$, aug-cc-pVDZ	6914.3	6140.6	17527.7	3242.5	3745.8	2084.4	3109.9	1339.3	3123.6	1316.9	3499.1	1546.7
C_8H_{10}, cc-pVTZ	2081.0	1877.0	6042.8	1208.3	1241.6	802.1	1022.5	493.5	1079.0	508.5	1287.9	686.0
$C_{12}H_{14}$, cc-pVTZ	14448.8	12220.4		6672.5	8325.0	4591.0	7299.6	3136.3	7345.6	3004.4	8659.6	3903.3

[a]Intel Core2 Quad Q9300 processor was used in our test calculation. [b]Efficiency fallback due the use of the same node in case of these examples. Original suggestion was to use MPI version of CFOUR on different nodes.[11] [c]The ladder type calculation use a single core, but other parts use four cores utilized by OpenMP parallelized GEMM. It is the default adjustment on a single node.

results in only a small (~10%) gain in the overall execution time. The reason for this can be best understood in Figure 5e: the matrix multiplication is much more effective than individual multiplication of the nonzero elements, and the performance (FLOP rate) decreases fast when more and more blocks are multiplied with the sparse algorithm. One cause for this fast decay is the inefficient use of the L2 cache (Figure 5f).

In conclusion, it seems that the execution time is not very sensitive to the actual value of the density criterion. Only blocks having no more than about 5−15% nonzero elements should be multiplied with sparse algorithm, in all other cases the full matrix−matrix multiplication is preferred due to the efficiency of this type of operations. In all of our further calculations, 15% will be used as density criterion, which results in about 10% gain of computer time.

Efficiency of the New CC Code. To test the real performance of the new algorithm, we have performed a series of calculations on conjugated polyenes. Execution time with different basis sets, different group sizes, and different number of cores have been compared. The results obtained for 1,3,5,7-octatetraene (C_8H_{10}), 1,3,5,7,9,11-dodecahexaene ($C_{12}H_{14}$), 1,3,5,7,9,11,13,15-hexadecaoctaene ($C_{16}H_{18}$), and 1,3,5,7,9,11,13,15,17,19-icosadecaene ($C_{20}H_{22}$) in the all-trans configurations are listed in Table 3. As a reference, the table also lists the timings for the original sparse matrix method of CFOUR, its MPI parallelization (on four cores) over integrals, and a version which uses trivial OpenMP parallelization over looping index C (see Scheme 1).

First, we try to obtain the optimal value for the orbital group size. As mentioned previously, both too small and too large orbital group sizes should decrease the efficiency of matrix multiplications. The results in Table 3 clearly show this: the orbital group sizes of 4 and 32 were too small and too large, respectively, to get optimal performance. It is also clear that the optimal value of the orbital group size depends on the average number of basis functions per atom. While for the cc-pVDZ basis the optimal orbital group size was found to be 8, for the larger aug-cc-pVDZ and cc-pVTZ basis sets both 8 and 16 were good choices. A speed-up of 4−6 is obtained when comparing with the trivial OpenMP version for the ladder type term and the gain is similar with respect to the original MPI parallelization in case of smaller systems. (The MPI version of CFOUR was designed for use on different nodes.[25] In case of these examples, it was used on the same node, and it has efficiency fallback for larger systems due to I/O bottleneck.)

Unfortunately, the speed-up gained for the ladder type contraction is suppressed by other contributions within the CC loop. In case of the $C_{12}H_{14}$ test molecule, we observe a total speed-up of 2 with the smaller cc-pVDZ basis set, 2.2 with the larger cc-pVTZ basis set, and 2.5 with the diffuse aug-cc-pVDZ

basis set, which was 4−4.5 for the ladder type contribution. The other than ladder terms scale with the lower power of the number of virtual orbitals, but the sparsity of the quantities involved is not used in the present version of our code. In particular, the growth of the molecule results in a rapid increase of the execution time of these contributions, mainly due to the increase of the number of occupied orbitals. In the present test calculations, a speed-up of 1.5 to 3 is observed, but it is assumed that it would decrease further with the size of the calculations. This observation warrants a closer investigation of these other terms, but this is out of the scope of the present paper. We note in passing that the speed-up of the entire CC loop is still 3.7−5.2 with respect to the serial version when compared with the calculation on four cores (group size 8).

These timings do not contain the time spent with the integral sorting because it is an additional step in the first iteration only. Still, in some cases, the time required for the sorting is not negligible; therefore, it is also listed in Table 3. For small molecules, the number of two-electron integrals is proportional to the fourth power of the number of basis functions, and consequently, the resource need for the integral sorting problem increases with a similar rate. On the other hand, the ladder type contraction and consequently the CC iteration scales with a higher power; therefore, the relative weight of the sorting in the total execution time decreases. In particular, the integral sorting for the C_8H_{10} molecule with cc-pVDZ basis set needs twice as much time as a complete CC iteration loop with the new algorithm, but this rate is only 1.7 with the aug-cc-pVDZ basis set and decreases to 1.4 in case of the cc-pVTZ basis set. By increasing the size of the molecule, this rate also decreases: for the $C_{20}H_{22}$ molecule, it is only 14% of a CC iteration loop. Note that the sorting time is always less than the time required for one iteration of the original procedure. Therefore, one can state that, although the integral sorting is an expensive part of the calculation, it is compensated multiple times by the speed-up of the new ladder type contraction.

As we were interested in comparing the performance of our algorithm to the theoretical performance limits of the computer, as well as to the performance achievable with the LINPACK package, we performed further benchmark calculations with the ladder-type multiplication code. Table 4 presents the results obtained for three test systems: cytosine, adenine, and 1,3,5,7,9,11-dodecahexaene ($C_{12}H_{14}$). Cytosine and adenine are "compact" molecules, while $C_{12}H_{14}$ is of "chain" type being more representative for sparsity. Results are given for two different block sizes, 8 and 16, and as well as for the original algorithm with the trivial OpenMP parallelization on four CPU cores. For completeness, the original MPI results (4 threads) are also given (note the limitation of this algorithm on one node). The theoretical peak performance (bottom row of the table) has been calculated from clock rate; note that 85% of this theoretical value could be measured by the LINPACK benchmark, and our plain multiplications also show about the same performance for large matrices (see Table 2). One observes that the measured performance is between 50 and 70% of the theoretical performance and between 60 and 85% of the LINPACK performance. FLOP rate grows with the number of occupied orbitals, that is, with the increasing number of multiplications. The measured performance also increases when molecular symmetry is ignored, and therefore, the amplitude matrix is larger. It also increases with the block size, which again results in larger amplitude matrix. This is in line with our findings in connection to Table 2, where we have seen that the speed-up of the multiplication increases with the matrix size. Nevertheless, the algorithm utilizes the hardware much better, resulting in a speed-up of 3 or more. The results also demonstrate that, with the optimal block size, the algorithm works equally well for compact and chain-type molecules.

Table 4. Results of the Performance Tests Obtained for the Ladder Type Contribution[a]

	N_{occ}	block size	DP performance[b] (GFLOP/s)
New, Using C_s Symmetry:			
cytosine, cc-pVTZ	29	8	19.6
adenine, cc-pVTZ	35	8	20.7
$C_{12}H_{14}$, cc-pVDZ	43	8	21.7
cytosine, cc-pVTZ	29	16	21.9
adenine, cc-pVTZ	35	16	22.3
$C_{12}H_{14}$, cc-pVDZ	43	16	27.6
New, Using C_1 Symmetry			
cytosine, cc-pVTZ	29	8	22.8
adenine, cc-pVTZ	35	8	22.3
cytosine, cc-pVTZ	29	16	23.4
adenine, cc-pVTZ	35	16	26.3
Original, MPI, Using C_s Symmetry			
cytosine, cc-pVTZ			4.5
adenine, cc-pVTZ			5.2
Trivial OMP, Using C_s Symmetry			
cytosine, cc-pVTZ			7.5
adenine, cc-pVTZ			7.9
LINPACK			32.5[c]
theoretical limit			38.4[d]

[a]Intel Core2 Quad Q9300 processor, four cores, OpenMP parallelization. The tested cycle contains the reading of the AO integrals, multiplication of the dense and sparse blocks, as well as the resorting of the amplitude and intermediate quantities. [b]Benchmarked with PAPI (Performance Application Programming Interface).[40] [c]Benchmarked with Intel MKL 11.1 LINPACK. [d]Calculated using the formula: GFLOP/s = 4 (number of core) × 2.4 GHz (frequency) × 4 double-precision FLOP/cycle.

In summary, the test results shows that the new algorithm is efficient for not too small systems with not too small basis sets. The optimal orbital group size was 8 in almost all cases, but it depends on the basis set size. In our experience, the cc-pVTZ and aug-cc-pVDZ basis sets were large enough to use a larger orbital group size (16), which leads to a higher FLOP rate. However, the optimal orbital group size also depends on the computer details, primarily on the number of processor cores.

CONCLUSIONS AND OUTLOOK

We have presented a new algorithm for the calculation of the ladder-type term of the CCSD method using the quantities in the AO basis. This new algorithm has several ingredients. First, we perform the matrix multiplications with small blocks instead of performing one huge multiplication. To reduce the number of multiplications, we use an orbital grouping method to separate the dense and sparse blocks of the integral matrix. The orbital grouping we suggest does not need any topological information about the molecule; rather, it uses a carefully designed target function to measure the efficiency of the separation. This is very advantageous for the user, since this step is completely black-box and can eventually be used to partiton integral matrices not only in AO basis but in other (e.g. localized orbital basis) as well. Dense blocks are treated in full matrix multiplications, while in the case of sparse blocks only

the nonzero elements are treated. We demonstrated that the multiplication of the dense blocks can be efficiently performed on parallel architectures and therefore, for the ladder-type term, a speed-up of 4−6 can be obtained against previous parallel implementations.

In principle, the algorithm could be used also in an integral direct fashion, since only a single block of integrals is needed for the multiplication at once. However, the calculation of the blocks obtained in our partitioning procedure might not be optimal from the point of view of the integral evaluation. This aspect needs further investigation.

The algorithm is also suitable for use on GPUs. First, by partitioning the matrices to smaller blocks, the memory of the GPU can be used efficiently even if the entire integral matrix is too large to fit in it. Second, the transfer of integrals to the memory of the graphics card, which is a potential bottleneck, is reduced by the in place generation of the redundant blocks. Third, the control over the block size provided by our algorithm allows the maximization of the efficiency by ensuring the full load of the computation device. Work in this direction is also in progress in our laboratory.

AUTHOR INFORMATION

Corresponding Author

*E-mail: szalay@chem.elte.hu.

Notes

The authors declare no competing financial interest.

ACKNOWLEDGMENTS

The authors acknowledge support by OTKA (Grant No. F72423). The European Union and the European Social Fund have provided financial support to the project under Grant No. TÁMOP 4.2.1./B-09/1/KMR-2010-0003.

REFERENCES

(1) Čížek, J. *J. Chem. Phys.* **1966**, *45*, 4256.

(2) Bartlett, R. J. *J. Chem. Phys.* **1989**, *93*, 1697.

(3) Lee, T. J.; Scuseria, G. E. In *Quantum Mechanical Electronic Structure Calculations with Chemical Accuracy*; Langhoff, S. R., Ed.; Kluwer Academic Publishers: Dordrecht, 1995; p 47.

(4) Gauss, J. In *Encyclopedia of Computational Chemistry*; Schleyer, P. v. R., Allinger, N. L., Clark, T., Gasteiger, J., Kollmann, P., Schaefer, H. F., Schreiner, P. R., Eds.; Wiley: Chichester, 1998; p 615.

(5) Bartlett, R. J.; Musial, M. *Rev. Mod. Phys.* **2007**, *79*, 291.

(6) Bartlett, R. J. *WIREs Comput. Mol. Sci.* **2012**, *2*, 126−138.

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

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

(9) Bartlett, R. J.; Watts, J. D.; Kucharski, S. A.; Noga, J. *Chem. Phys. Lett.* **1990**, *165*, 513.

(10) Stanton, J. F.; Gauss, J.; Watts, J. D.; Lauderdale, W. J.; Bartlett, R. J. *Int. J. Quantum Chem.* **1992**, *26*, 879.

(11) CFOUR, a quantum chemical program package written by Stanton, J. F., Gauss, J., Harding, M. E., Szalay, P. G. with contributions from Auer, A. A.; Bartlett, R. J.; Benedikt, U.; Berger, C.; Bernholdt, D. E.; Bomble, Y. J.; Cheng, L.; Christiansen, O.; Heckert, M.; Heun, O.; Huber, C.; Jagau, T. C.; Jonsson, D.; Jusélius, J.; Klein, K.; Lauderdale, W. J.; Matthews, D. A.; Metzroth, T.; Mück, L. A.; O'Neill, D. P.; Price, D. R.; Prochnow, E.; Puzzarini, C.; Ruud, K.; Schiffmann, F.; Schwalbach, W.; Stopkowicz, S.; Tajti, A.; Vázquez, J.; Wang, F.; Watts, J. D.; and the integral packages MOLECULE (Almlöf, J.; Taylor, P. R.), PROPS (Taylor, P. R.), ABACUS (Helgaker, T.; Jensen, H. J. Aa.; Jørgensen, P.; Olsen, J.), and ECP routines by Mitin, A. V.; van Wüllen, C. http://www.cfour.de (accessed April 23, 2012).

(12) Werner, H.-J.; Knowles, P. J.; Knizia, G.; Manby, F, R.; Schütz, M. *WIREs Comput. Mol. Sci.* **2012**, *2*, 242−253.

(13) Aquilante, F.; Vico, L. D.; Ferré, N.; Ghigo, G.; Malmqvist, P.; Neogrády, P.; Pedersen, T. B.; Pitoňák, M.; Reiher, M.; Roos, B. O.; Serrano-Andrés, L.; Urban, M.; Veryazov, V.; Lindh, R. *J. Comput. Chem.* **2010**, *31*, 224−247.

(14) Baker, J.; Janowski, T.; Wolinski, K.; Pulay, P. *WIREs Comput. Mol. Sci.* **2012**, *2*, 63−72.

(15) Crawford, T. D.; Sherrill, C. D.; Valeev, E. F.; Fermann, J. T.; King, R. A.; Leininger, M. L.; Brown, S. T.; Janssen, C. L.; Seidl, E. T.; Kenny, J. P.; Allen, W. D. *J. Comput. Chem.* **2007**, *28*, 1610−1616.

(16) Kállay, M. *MRCC, A String-Based Quantum Chemical Program Suite*. http://www.mrcc.hu (accessed April 23, 2012).

(17) Kállay, M.; Surján, P. R. *J. Chem. Phys.* **2001**, *115*, 2945.

(18) van Dam, H. J. J.; de Jong, W. A.; Bylaska, E.; Govind, N.; Kowalski, K.; Straatsma, T. P.; Valiev, M. *WIREs Comput. Mol. Sci.* **2011**, *1*, 888−894.

(19) Deumens, E.; Lotrich, V. F.; Perera, A.; Ponton, M. J.; Sanders, B. A.; Bartlett, R. J. *WIREs Comput. Mol. Sci.* **2011**, *1*, 895−901.

(20) Hampel, C.; Peterson, K. A.; Werner, H.-J. *Chem. Phys. Lett.* **1992**, *1*, 190.

(21) Schutz, M.; Lindh, R.; Werner, H.-J. *Mol. Phys.* **1999**, *96*, 719−733.

(22) Kobayashi, R.; Rendell, A. P. *Chem. Phys. Lett.* **1997**, *265*, 1−11.

(23) Janowski, T.; Pulay, P. *J. Chem. Theory Comput.* **2008**, *4*, 1585−1592.

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

(25) Harding, M. E.; Metzroth, T.; Gauss, J. *J. Chem. Theory Comput.* **2008**, *4*, 64−74.

(26) Lotrich, V.; Flocke, N.; Ponton, M. J.; Yau, A.; Perera, A.; Deumens, E.; Bartlett, R. J. *J. Chem. Phys.* **2008**, *128*, 194104.

(27) Haser, M.; Ahlrichs, R. *J. Comput. Chem.* **1989**, *10*, 104−111.

(28) Lawson, C.; Hanson, R.; Kincaid, D.; Krough, F. *ACM Trans. Math. Software* **1979**, *5*, 308−325.

(29) Borkar, S. Thousand core chips: A technology perspective. *Proceedings of the 44th Design Automation Conference*, San Diego, CA, June 4−8, 2007.

(30) Challacombe, M. *J. Chem. Phys.* **1999**, *110*, 2332.

(31) Saravanan, C.; Shao, Y.; Baer, R.; Ross, P. N.; Head-Gordon, M. *J. Comput. Chem.* **2003**, *24*, 618−622.

(32) Dongarra, J.; DuCroz, J.; Duff, I.; Hammarling, S. *ACM Trans. Math. Software* **1989**, *16*, 1−17.

(33) Çatalyürek, Ü. V.; Aykanat, C.; Uçar, B. *SIAM J. Sci. Comp.* **2010**, *32*, 656−683.

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

(35) Karypis, G.; Kumar, V. *METIS, A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*. http://glaros.dtc.umn.edu/gkhome/metis/metis/download (accessed April 23, 2012).

(36) Dunning, T. H. *J. Chem. Phys.* **1989**, *90*, 1007−1023.

(37) Kendall, R. A.; Dunning, T. H.; Harrison, R. J. *J. Chem. Phys.* **1992**, *96*, 6796−6806.

(38) The MPIForum. MPI: A message passing interface. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*; ACM Press: Portland, OR, 1993.

(39) Dagum, L.; Menon, R. *IEEE Comput. Sci. Eng.* **1998**, *5*, 46−55.

(40) Mucci, P. J. *PAPI, A Portable Interface to Hardware Performance Counters on Microprocessors*. http://icl.cs.utk.edu/papi/software/index.html (accessed Aug. 1, 2012).