Cleaning up redundant zeroes in RHF CCSD amplitudes, intermediates and double bar integrals

Ole Tobias Norli

Abstract In this letter we present a new solution to redundant (zero) values in double bar integrals, intermediates and amplitudes for RHF CCSD calculations. The solution is very simple rearrangement of orbitals in terms of indexing. The result is a 40 % faster code and more than 50 % reduction in memory needs, compared to keeping the zeroes. [Also a series of other optimizations already implemented will be described in futures editions. Program uses MPI for parallel.]

Keywords RHF CCSD · Factorization · Compact storage · Double Bar Integrals · EMSL

1 Introduction

Coupled Cluster Singles and Doubles, CCSD, is widely recognized as one of the more accurate methods in electron structure theory. Descriptive articles in this field include J. F. Stanton et al. (1991), Henrik Koch et al. (1994), P. Carsky et al. (1987) and Schaefer (2009). Newer articles include Jonathan L. Bentz et al. (2007). There are several challenges when constructing an optimized CCSD code. In an unoptimized code the largest calculation scales as $n_v^4 n_o^2$, where n_v is the number of virtual orbitals and n_o is the number of occupied orbitals. The largest array needed for storage in an unoptimized code scales as n_v^4. This largest variable is a result from transforming the two electron integrals and then forming the so called double bar integrals. These however are diagonal in total spin projection. This will mean an unoptimized form of the double bar integrals will be filled with approximately 50 % of the terms being equal to zero. Prior to this article there have been several solutions proposed, but most of them are complex. One of the more common is a *block diagonalization*. We have studied an RHF based CCSD. In this letter we will propose a new and extremely easy method, that has been heretofore unmentioned in the literature, of removing the zero terms without any complications.

In the letter we will first list our factorization, then discuss the problem in greater detail, and then propose our solution. We finally conduct testing to see how much improvement is achieved.

This is a first and temporary edition of an article. Comments of content to be better described in future editions is added inside brackets like this: [More description coming here about this, etc].

2 Factorization

Our CCSD factorization is based on the work by E. Solomonik et al. (2011) and references therein. The notation v_{ab}^{ij} is used for $\langle ab||ij\rangle$. Using i,j,k,l and (a,b,c,d) for occupied and (unoccupied) orbitals in RHF the CCSD equations can be written as such

$$E_{CCSD} = \sum_{ai} f_a^i t_i^a + \frac{1}{4}\sum_{abij} v_{ij}^{ab}\tau_{ij}^{ab} \tag{1}$$

$$\begin{aligned} t_i^a D_i^a =& f_i^a - \sum_k t_k^a [F_2]_i^k + \sum_{c\neq a} f_c^a t_i^c + \sum_{kc} v_{ka}^{ci} t_k^c \\ & - \frac{1}{2}\sum_{klc} [W_3]_{kl}^{ci} t_{kl}^{ca} + \frac{1}{2}\sum_{kcd} v_{ka}^{cd}\tau_{ki}^{cd} + \sum_{kc} t_{ik}^{ac}[F_1]_c^k \end{aligned} \tag{2}$$

Ole Tobias Norli

$$t_{ij}^{ab} D_{ij}^{ab} = v_{ij}^{ab} + \frac{1}{2}\sum_{kl} \tau_{kl}^{ab} [W_1]_{ij}^{kl} - \mathbf{P}(ab) t_k^b [W_2]_{ij}^{ak} + \mathbf{P}(ij) \sum_k t_{jk}^{ab} [F_2]_i^k + \frac{1}{2}\sum_{cd} v_{ab}^{cd} \tau_{ij}^{cd} + \mathbf{P}(ab) \sum_c t_{ij}^{bc} [F_3]_c^a + \mathbf{P}(ij) \sum_c v_{ab}^{cj} t_i^c + \mathbf{P}(ab)\mathbf{P}(ij) \sum_{kc} t_{jk}^{bc} [W_4]_{ic}^{ak} \quad (3)$$

Where Eqs. (4) to (13) defines the intermediates.

$$[F_1]_c^k = f_c^k + \sum_{cd} v_{kl}^{ld} t_l^d \quad (4)$$

$$[F_2]_i^k = (1 - \delta_{ki})\, f_i^k + \sum_c t_i^c [F_1]_c^k + \sum_{cl} v_{kl}^{ic} t_l^c + \frac{1}{2}\sum_{cld} v_{kl}^{cd} t_{il}^{cd} \quad (5)$$

$$[F_3]_c^a = (1 - \delta_{ac})\, f_c^a - \sum_k t_k^a [F_1]_c^k - \sum_{kd} v_{ka}^{cd} t_k^d + \frac{1}{2}\sum_{kld} v_{kl}^{cd} t_{kl}^{ad} \quad (6)$$

$$[W_1]_{ij}^{kl} = v_{kl}^{ij} + \sum_c \mathbf{P}(ij) v_{kl}^{cj} t_i^c + \frac{1}{2}\sum_{cd} v_{kl}^{cd} \tau_{ij}^{cd} \quad (7)$$

$$[W_2]_{ij}^{ak} = v_{ak}^{ij} + \sum_c \mathbf{P}(ij) v_{ak}^{ic} t_j^c + \frac{1}{2}\sum_{cd} v_{ak}^{cd} \tau_{ij}^{cd} \quad (8)$$

$$[W_3]_{ci}^{kl} = v_{kl}^{ci} + \sum_d v_{kl}^{cd} t_i^c \quad (9)$$

$$[W_4]_{ic}^{ak} = v_{ak}^{ic} + \sum_d v_{ak}^{dc} t_i^d - \sum_l t_l^a [W_3]_{ci}^{kl} + \frac{1}{2}\sum_{ld} v_{kl}^{cd} t_{il}^{ad} \quad (10)$$

$$\tau_{ij}^{ab} = t_{ij}^{ab} + t_i^a t_j^b - t_j^a t_i^b \quad (11)$$

$$D_i^a = f_i^i - f_a^a \quad (12)$$

$$D_{ij}^{ab} = f_i^i + f_j^j - f_a^a - f_b^b \quad (13)$$

This factorization ensures that all terms can be calculated using external math libraries. Also the largest intermediate scales as $n_v^2 n_o^2$ in terms of memory requirements.

3 Compact storage

We now consider the molecular orbitals, which are calculated as such

$$\langle pq|rs\rangle = \sum_{\alpha\beta\xi\nu} C_\alpha^p C_\beta^q C_\xi^r C_\nu^s \langle \alpha\beta|\xi\nu\rangle \quad (14)$$

Here $\langle \alpha\beta|\xi\nu\rangle$ are our atomic orbitals (AOs). These come from our RHF calculations. $\langle pq|rs\rangle$ are the molecular orbitals (MOs). MOs here are presented as a linear combination of AOs. It should be noted the symmetries from the AOs are kept in the MOs. The MOs appear in CCSD as a double bar integral. This is defined as such

$$\langle pq||rs\rangle = \langle pq|rs\rangle - \langle pq|sr\rangle \quad (15)$$

Due to spin considerations, if we fill a matrix with $\langle pq||rs\rangle$ it will be filled with mostly zeroes. However when using an RHF based CCSD it is common that all even numbered orbitals have the same spin orientation. This means all odd numbered orbitals will also have the same spin orientation. This results in the zeroes forming pattern that has now been identified and utilized.

$\langle pq||rs\rangle$ are diagonal in total spin projection. In RHF the total spin is also equal to zero. When we have all odd numbered orbitals with the same spin orientation, and same with even numbered orbitals, this has a practical implication. The implication is that the only terms that will not be equal to zero are those where the sum of the orbital indexes are equal to an even number.

We will now visualize this. We construct a program that performs the AO to MO transformation and print $\langle pq||rs\rangle$ for a fixed $p = 1$ and $r = 1$. In the span of q and s there is formed a matrix, we have noted the

terms that will be zero and also the terms that will be non-zero with the indexes (q, s).

$$\begin{pmatrix} (0,0) & 0 & (0,2) & 0 & (0,4) & 0 & \dots \\ 0 & (1,1) & 0 & (1,3) & 0 & (1,5) & \dots \\ (2,0) & 0 & (2,2) & 0 & (2,4) & 0 & \dots \\ 0 & (3,1) & 0 & (3,3) & 0 & (3,5) & \dots \\ (4,0) & 0 & (4,2) & 0 & (4,4) & 0 & \dots \\ 0 & (5,1) & 0 & (5,3) & 0 & (5,5) & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \end{pmatrix}$$

This array is now stored in our computer as a four dimensional array that we call $I[a][b][c][d]$. We on purpose use a different kind of indexing for our stored integrals, even though index a in this example is a referance to orbital p. We can note which orbitals our array-indexes refers to as such

a = p
b = r
c = q
d = s

This will be an array of size $(2N)^4$, where N is the number of contraction Gaussian Type Orbitals (GTOs). We now perform a trick. We want our indexes of I to refer to a different orbital, in practise we want:

a = p
b = r
c = q/2 + (q% 2) N
d = s/2 + (s% 2) N

Where % is the binary operator and we use integer division by 2. Now index c is no longer a referance to orbital q, but a referance to orbital [q/2 + (q % 2) N]. If we now visualize the same double bar integral with fixed $p = 1$ and $r = 1$ it looks like this

$$\begin{pmatrix} (0,0) & (0,2) & (0,4) & \dots & 0 & 0 & 0 & \dots \\ (2,0) & (2,2) & (2,4) & \dots & 0 & 0 & 0 & \dots \\ (4,0) & (4,2) & (4,4) & \dots & 0 & 0 & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ (1,1) & (1,3) & (1,5) & \dots & 0 & 0 & 0 & \dots \\ (3,1) & (3,3) & (3,5) & \dots & 0 & 0 & 0 & \dots \\ (5,1) & (5,3) & (5,5) & \dots & 0 & 0 & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \end{pmatrix}$$

Performing this trick will always result in a matrix that looks something like this. We can split this matrix into four sub-matrices, one top left, one top right, one bottom left and one bottom right. Regardless of a and b, we will always have either the two left sub-matrices, or the two right sub-matrices always filled with zeroes. These do not need to be stored. If we ensure we *only perform calculations on orbitals with a non-zero contribution* we can change our array-indexing to:

a = p
b = r
c = q/2 + (q% 2) N
d = s/2

And also reduce the size of the array by half. Visualizing now the same array it looks like this.

$$\begin{pmatrix} (0,0) & (0,2) & (0,4) & \dots \\ (2,0) & (2,2) & (2,4) & \dots \\ (4,0) & (4,2) & (4,4) & \dots \\ \dots & \dots & \dots & \dots \\ (1,1) & (1,3) & (1,5) & \dots \\ (3,1) & (3,3) & (3,5) & \dots \\ (5,1) & (5,3) & (5,5) & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix}$$

And its size will be $\frac{1}{2}(2N)^2$. This kind of indexing can and should be performed on **all** stored integrals, amplitudes and intermediates. This ensures all memory is reduced by at least 50 %. Also if a,b,c,d is referencing orbitals in the same manner in all stored arrays we can still use external math libraries as before. However now we will not be passing any zeroes into these external math libraries, so calculations can be faster. This change in indexing keeps all symmetries and also allow easy row and column access. The row is accessed as usual.

$$\begin{pmatrix} (0,0) & (0,2) & (0,4) & \dots \\ (2,0) & (2,2) & (2,4) & \dots \\ (4,0) & (4,2) & (4,4) & \dots \\ \dots & \dots & \dots & \dots \\ (1,1) & (1,3) & (1,5) & \dots \\ (3,1) & (3,3) & (3,5) & \dots \\ (5,1) & (5,3) & (5,5) & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix}$$

A column is slightly different, since we only require either the top half or the bottom half of the matrix.

$$\begin{pmatrix} (0,0) & (0,2) & (0,4) & \dots \\ (2,0) & (2,2) & (2,4) & \dots \\ (4,0) & (4,2) & (4,4) & \dots \\ \dots & \dots & \dots & \dots \\ (1,1) & (1,3) & (1,5) & \dots \\ (3,1) & (3,3) & (3,5) & \dots \\ (5,1) & (5,3) & (5,5) & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix}$$

The attributes of the double bar integrals in RHF are such that there will be some remaining zeroes after removing these. However there will be another pattern formed where all remaining zeroes are placed in one

of two remaining sub-matrices. This can also be accounted for, reducing memory needs by an additional $\frac{1}{8}$. [Mention that this is strictly in RHF].

It should be mentioned that even when reducing the memory requirements by roughly 50 % it is not enough to run large systems. The problem with memory is a scaling problem, hence additional optimizations should be performed. The material presented here can be integrated very easily however you wish to optimize your code.

[Storing in this manner leads directly to another very important optimization. When factorizing our CCSD equations we are vulnerable to multiplying terms together that will later be multiplied by zero, and the CPU time used will be wasted. Since we arrange our orbitals as odd and even we can split the sums into odd and even combinations. For example t_{ij}^{ab} amplitude can be split into (odd a, odd b, odd i, odd j), (even a, odd b, even i, odd j) etc. This ensures we do not need to perform % calculations in our code. We can also very easily identify what terms that will later be multiplied by zero. More description here, this gave big speed-up.]

4 Parallel implementation

For parallel implementation we will use MPI. We first study one specific variable, $t2new$. This variable will contain the new T2 amplitudes and scales as $n_v^2 n_o^2$. For parallel implementation we will remove this variable from storage and replace it with two one dimensional arrays. These two one dimensional arrays are set in size prior to iterating and will be of size $\frac{n_v^2 n_o^2}{M}$, where M is the number of procs. One of the arrays will hold the calculations performed by the core itself, the other array will be information communicated to the core.

What we do is map out in a cyclical manner the work to be done by each core. This work is then performed and stored in the first array. Afterwards there is called a new function that maps the new amplitudes into the $t2$ array, that contains the current T2 amplitudes. At this point there are no more calculations to be done so we can overwrite the old amplitudes. We then loop over the number of cores, find out what work the current core has performed, have this core communicate the information into the second stored array at all cores and map it into t2. Then we repeat this procedure for the next core and overwrite the information already mapped. This reduce our memory needs considerably.

We can also use the same two arrays when running everything else in parallel, and simply replace the content of the arrays. This ensures we can do parallel and at the same time reduce our memory needs, as the new T2 amplitudes are only needed at the very end of an iteration.

The current parallel implementation is implemented with significant memory distribution. All arrays scaling higher than $n_v^2 n_o^2$ are distributed among nodes. We have also kept the option of storing to disk easily available, this is however not utilized currently. [AO->MO transformation does not run in parallel currently, it will soon.] The integrals stored are singel bar integrals, however these are split up into smaller chunks. The large chunks are distributed in memory as done in HF.

5 Results

For testing we have constructed two programs that are identical in all ways but one. The first program uses the normal storage of zeroes and everything, and the second program uses the compact storage. Also different is how symmetry, rows and columns are accessed, and a slight difference in how external math libraries are initialized. It is already obvious that the second program uses half the memory of the first, but we have tested the speed of the two implementations. We first want to benchmark our programs. For this we plot the energy as a function of R between two small molecules. We use the basis set 6-311ss.

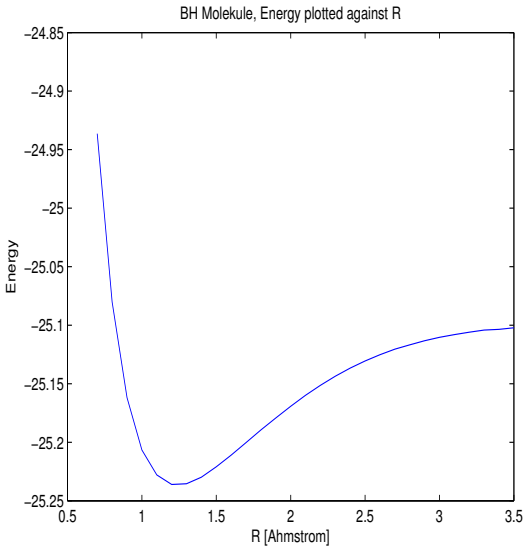

Fig. 1 Energy of BH molecule as a function of R.

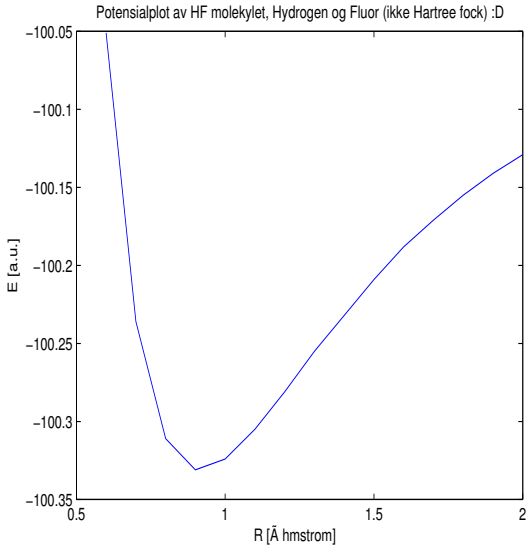

Fig. 2 Energy of HF molecules as a function of R.

These calculations are benchmarked against Arteum D. Bochevarov and C. David Sherrill (2005). We also do calculations for the water molecule with the $STO-3G$ basis set. The CCSD correction energy E_{corr} = -0.0501273 and the total energy E_{CCSD} = -75.0128. These are benchmarked against NWChem down to the final decimal. The number of iterations was 20, this is also benchmarked against NWChem. Additional benchmarks performed against LSDALTON.

[Currently small bug in BOYS function. This occurs only when high angular momentum orbitals are present and is present as a systemic bug which shifts our energies approximatly 0.1 %. The shape is therefore identical to benchmark, so bug is hard to see if only looking at graphs. Smaller angular momentum CCSD gives energies identical to benchmarks (LSDALTON and NWChem) down to the final decimal. Bug very fixable, much work done to identify.]

We then do calculations for 10 electrons with different basis sets from EMSL Basis Set Exchange, EMSL (2007). We use a single threaded 2000 MHz CPU with 8192 RAM. Listed are number of occupied orbitals, number of virtual orbitals, total time measured with normal storage and total time measured with compact storage. Convergence criteria was set to 10^{-8} a.u. and convergence is defined when the difference in energy from one iteration to the next is smaller than this value. n_o is always equal to 10.

n_v	$Normal[s]$	$Compact[s]$	$Compact/Normal$
16	0.18	0.11	0.61
28	0.93	0.54	0.58
40	3.35	2.06	0.61
52	7.43	4.47	0.60
82	46.36	28.05	0.60
120	171.18	99.60	0.58

The program is available in github (2013). All energies and number of iterations were identical from both codes. For our intermediate $[W_3]$ we also make c and d dependant upon a and b, to remove the additional $\frac{1}{8}$ of memory.

The time per iteration for $n_o = 10$ and $n_v = 120$ was 2.45 seconds. The available code has later been implemented in parallel using MPI and further optimized.

[Rest of result section must be revised]
Calculations using 6-311(2d,2p) on one water molecule have been done and the time per CCSD iteration compared to LSDALTON. The program for this system does one iteration in 0.82/0.83s. This is measured on my office computer to be twice as fast as LSDALTON (slightly better than twice as fast, but LSDALTON used DIIS or something similar which gives slight increase in time per iteration). So far unable to measure time per iteration in LSDALTON, however speed were measured by running both codes at the same time and noting two iterations or more every time one LSDALTON iteration finished. Input file used in LSDALTON was:
**WAVE FUNCTIONS
.HF
**CC
.MEMORY
8
.CCSD
.CCSDNOSAFE
*END OF INPUT
An effort was made to supply more than enough RAM. In parallel implementation two grids are used, one of size n_v^2 and one of size $n_v n_o$, there are three places of communication with everything going in parallel except for the Energy calculation (n^4) and the calculation of τ_{ij}^{ab} (n^4). The grids are distributed block-cyclic, with block size = 2, to utilize the advantages from compact storage. This ensures we get one (odd-odd-odd-odd), one (even-odd-eve-odd) etc.. combinations in each block. A purely cyclic approach is also possible and easily implemented, if it shows to be more effective. A block cyclic approach also allows storage and distribution of single bar integrals, and then map these into double bar integrals. This enables us to avoid much

redundant storage. This mapping was used and slowed down our calculations when comparing to LSDALTON for the small system. One redudant loop had to be used in the mapping of one of the MOs. Listed now are the parallel results using the same system for increasing nr of cores. First column nr of CPUs, second time per iteration, third the mean time per iteration and last the time per iteration for the nr of procs used divided by time per iteration for one proc.

$cpus$	$t/iter(varies)$	$Mean$	$t(procs)/t(1proc)$
1	$0.82 - 0.83$	0.82	1
2	$0.43 - 0.44$	0.43	0.524
4	$0.26 - 0.28$	0.27	0.329
6	$0.17 - 0.20$	0.18	0.220
8	$0.15 - 0.17$	0.16	0.195

6 Conclusion

We see a roughly 40 % faster code with compact storage. External math libraries such as BLAS are known to perform better with increasing matrix size. Since our matrices are now smaller in size these math libraries will not perform as efficient. However the problems they will solve are half the size, so they will be solved faster. Matrices can be split into two and they will then be symmetric across the diagonal, however testing so far indicate it is faster to make the matrix as large as possible but also not store zeroes.

One major advantage is the simplicity of the solution. It is very easily implemented, and can be combined with other optimizations. The authors consider it amazingly exciting how these considerations would work with other implementations. One implementation in CCSD is available as noted previously on github (2013).

Compact storage can be applied to any method where double bar integrals are present, this includes but is not limited to CCSDT and CCSDTQ.

Nevin Oliphant (1991) noted in his PhD thesis the problem of redundant (zero) T2 amplitudes in MR-CCSD. The authors of this letter are curious if these considerations could be applied here.

[Parallel implementation gives decent results on the small number of CPUs used in the testing supplied here. Program has been mounted on abel, however there is a slight problem with the inclusion of MKL with armadillo. Currently a wrapper is used which slows down calculations. This will be solved as nr. 2 priority. RHF utilize a memory distribution solution for the two electron integrals and can run systems large enough to test the limits of the CCSD implementation. RHF has been tested for $n_o = 144$ and $n_v \approx 700$. The final goal is to run $n_o = 200$ and $n_v = 800$ to compare with the CTF code from Berkley. Program is already faster in serial but the genious of CTF is purely its parallel implementation and amazing scaling with increased nr of CPUs. Our factorization is based on CTF for this reason. Will probably need some more optimizations here.]

References

John F. Stanton, Jürgen Gauss, John D. Watts, and Rodney J. Bartlett, *A direct product decomposition approach for symmetry exploitation in many-body methods. I. Energy calculations* Chem. Phys. 94 (6), 15 March 1991

P. Carsky, L. J. Schaad, B. A. Hess, M. Urban, and J. Noga, *Use of molecular symmetry in coupled-cluster theory* J. Chem. Phys. 87,411 (1987).

Henrik Koch, Ove Christiansen, Rika Kobayashi, Poul Jorgensen , Trygve Helgaker *A direct atomic orbital driven implementation of the coupled cluster singles and doubles (CCSD) model* Chemical Physics Letters 228 (1994) 233-238

Edgar Solomonik, Devin Matthews, Jeff Hammond, James Demmel: Cyclops Tensor Framework: reducing communication and eliminating load imbalance in massively parallel contractions

The Role of Databases in Support of Computational Chemistry Calculations Feller, D., J. Comp. Chem., 17(13), 1571-1586, 1996

Basis Set Exchange: A Community Database for Computational Sciences Schuchardt, K.L., Didier, B.T., Elsethagen, T., Sun, L., Gurumoorthi, V., Chase, J., Li, J., and Windus, T.L. J. Chem. Inf. Model., 47(3), 1045-1052, 2007, doi:10.1021/ci600510j

An Introduction to Coupled Cluster Theory for computational Chemists, T. Daniel Crawford and Henry F. Schaefer. Center for Computational Quantum Chemistry, Department of Chemistry, The University of Georgia, Athens, Georgia 30602-2525.

Ole Norli *CCSD Program* Available on: *github.com*, user: *otnorli*, repository: *CCSD_PARALLEL*

Nevin Oliphant *A multi-reference coupled-cluster method using a single-reference formalism* PhD Thesis University of Arizona (1991)

Arteum D. Bochevarov and C. David Sherrill (1995). *Hybrid correlation models based on active-space partitioning: Correcting second-order Moller-Plesset perturbation theory for bond-breaking reactions* THE JOURNAL OF CHEMICAL PHYSICS 122, 234110 (2005)

Jonathan L. Bentz, RyanM.Olson, Mark S. Gordon, Michael W. Schmidt,Ricky A. Kendall *Coupled cluster algorithms for networks of shared memory parallel processors* Computer Physics Communications 176 (2007) 589-600