Another CCSD(T) Implementation

Andrey Asadchev Mark S. Gordon

Abstract

A new coupled cluster singles and doubles with triples correction, CCSD(T), algorithm is presented. The new algorithm is implemented in C++, has low memory footprint, fast execution time, low I/O overhead, and flexible storage backend with the ability to use either distributed memory or filesystem for storage. The algorithm is demonstrated to work well on single workstation, small cluster, and a high-end Cray computer. With the new implementation, a CCSD(T) calculation with several hundred basis functions and a few dozen occupied orbitals can run in under a day on a single workstation.

1 Introduction

As a rule of thumb, the electronic energy obtained through Hartree-Fock method accounts for the 99 % of the energy. However, the chemical properties of interest are dependent on the remaining 1 %. The difference between the reference Hartree-Fock energy and the true energy is defined as correlation energy,

$$E_{corr} = E_{hf} - E$$

Of the many electron correlation methods [1, 2, 3], the coupled cluster method with perturbative triples correction [4], CCSD(T), is the most successful, often referred to as the gold standard of the computational chemistry. The coupled cluster method was first developed by nuclear physicists [5], adopted to quantum chemistry by Cizek [6, 7, 8, 9, 10, 11], and popularized by Bartlett [12].

The coupled cluster method usually introduced in the exponential *ansatz* form [13, 14],

$$\Psi = e^T \Psi_0 = e^{(T_1+T_2+...T_n)}\Psi_0$$

where $T_1...T_n$ are the n-particle cluster operator and Ψ_0 is the reference wavefunction, typically the Hartree-Fock reference Ψ_{hf}.

The excitation operator applied to a reference wavefunction is written in terms of cluster excitation amplitudes t from hole states $i, j, k, ...$ (occupied orbitals in chemistry parlance) to particle states (or virtual orbitals), $a, b, c, ...$

$$T_n\Psi_0 = \sum_{ijk...}\sum_{abc...} t_{ijk...}^{abc...}\Psi_{ijk...}^{abc...}$$

Truncating the expansion at doubles, leads to coupled cluster singles and double method, CCSD,

$$T \approx T_1 + T_2$$

The t_i^a and t_{ij}^{ab} are found by solving a system of nonlinear equations,

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

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

where $H_N = H - \langle \Phi | H | \Phi \rangle$ is the normal order Hamiltonian [15]

The final algebraic CC equations, derived using diagramatic approach, result in a number of integral terms V contracted with T amplitudes, e.g. VT_1^2 signifies integral terms contracted with $t_i^a t_j^b$. The complete derivation can be found in the number of sources [16]. For the purposes of this work, the spin-free equations by Piecuch [17] are used.

As is customary, the algebraic CC equations are presented in Eistein summation, where repeated co- and contra-variant index implies summation. In terms of one-electron integrals $f_q^p = \langle p | f | q \rangle$ and two-electron molecular integrals v, the CCSD non-linear equations are,

$$\begin{aligned} D_i^a t_i^a = {} & f_i^a + I_e^a t_i^e - I'^m_i t_m^a + I_e^m (2t_{mi}^{ea} - t_{im}^{ea}) + t_m^e (2v_{ei}^{ma} - v_{ei}^{am}) \\ & v_{ei}^{mn} (2t_{mn}^{ea} - t_{mn}^{ae}) + v_{ef}^{ma} (2t_{mi}^{ef} - t_{im}^{ef}) \end{aligned} \tag{1}$$

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

$$I_a^i = f_a^i + 2v_{ae}^{im} t_m^e - v_{ea}^{im} t_m^e \tag{3}$$

$$I_b^a = (1 - \lambda_a^b) f_b^a + (2v_{be}^{am} t_m^e - v_{be}^{ma} t_m^e) - (2v_{eb}^{mn} c_{mn}^{ea} - v_{be}^{mn} c_{mn}^{ea}) - t_m^a f_b^m \tag{4}$$

$$I_j^i = I'^i_j + I_e^i t_j^e \tag{5}$$

$$I'^i_j = (1 - \lambda_j^i) f_j^i + (2v_{je}^{im} t_m^e - v_{ej}^{im} t_m^e) + (2v_{ef}^{mi} t_{mj}^{ef} - v_{ef}^{im} t_{mj}^{ef}) \tag{6}$$

$$I_{kl}^{ij} = v_{kl}^{ij} + v_{ef}^{ij} c_{kl}^{ef} + P(ik/jl) t_k^e v_{el}^{ij} \tag{7}$$

$$I_{jb}^{ia} = v_{jb}^{ia} - \frac{1}{2} v_{eb}^{im} c_{jm}^{ea} - v_{jb}^{im} t_m^a + v_{eb}^{ia} t_j^e \tag{8}$$

$$I'^{ab}_{ci} = v_{ci}^{ab} - v_{ci}^{am} t_m^b - t_m^a v_{ci}^{mb} \tag{9}$$

$$I'^{ia}_{jk} = v_{jk}^{ia} + v_{ef}^{ia} t_{jk}^{ef} + t_j^e t_k^f v_{ef}^{ia} \tag{10}$$

where D are the many-body denominators

$$D_{q\ldots}^{p\ldots} = f_q^q - f_p^p + \ldots$$

and P is the permutation operator,

$$P(ia/jb) u_{ab}^{ij} = u_{ab}^{ij} + u_{ba}^{ji}$$

Note that the permutation operator, akin to $A + A^{'}$, has the effect of symmetrizing the operand such that

$$P(ia/jb) u_{ab}^{ij} = P(ia/jb) u_{ba}^{ji}$$

The molecular integrals are obtained from atomic orbital (AO) basis via 4-index transformation,

$$v_{cd}^{ab} = C_p^a C_r^b C_c^q C_d^s \langle pq|\frac{1}{r}|rs\rangle$$

The transformed integrals have the following general symmetries,

$$v_{ar}^{bs} = v_{br}^{as}$$

$$v_{ab}^{qs} = v_{ba}^{sq}$$

The (T) correction is given as:

$$E^{[T]} = \bar{t}_{abc}^{ijk} t_{ijk}^{abc} D_{ijk}^{abc} \tag{11}$$

$$E^{(T)} = E^{[T]} + t_{ijk}^{abc} D_{ijk}^{abc} \bar{z}_{abc}^{ijk} \tag{12}$$

where

$$\bar{x}_{abc}^{ijk} = \frac{4}{3} x_{abc}^{ijk} - 2x_{acb}^{ijk} + \frac{2}{3} x_{bca}^{ijk}$$

$$z_{abc}^{ijk} = (t_a^i v_{bc}^{jk} + t_b^j v_{ac}^{ik} + t_c^k v_a^{ij} b)/D_{ijk}^{abc}$$

and the T_3 amplitudes are,

$$D_{ijk}^{abc} t_{ijk}^{abc} = P(ia/jb/kc)[t_{ij}^{ae} v_{ek}^{bc} - t_{im}^{ab} v_{jk}^{mc}] \tag{13}$$

where,

$$P(ia/jb/kc) u_{abc}^{ijk} = u_{abc}^{ijk} + u_{acb}^{ikj} + u_{bac}^{jik} + u_{bca}^{jki} + u_{cba}^{kji} + u_{cab}^{kij}$$

1.1 Computational details

CCSD equations are non-linear and must be solved to self-consistenct via iterative procedure, usually with a help of acceleration method [18]. The CCSD method is dominated by its most expensive term, $v_{ef}^{ab} c_{ij}^{ef}$, which scales as $v^4 o^2$. The algorithm is expensive in terms of memory and storage as well, with amplitude storage on the order of $v^2 o^2$ and integrals storage on the order of $v^4, v^3 o, ...$ and so on. The amount of the in-core memory depends on the algorithm; most of algorithms require $v^2 o^2$ storage per node. This amount of memory is non scalable, e.g. 100 occupied and 1000 virtual orbitals require 80GB of memory per node which is very uncommon.

The (T) is non-iterative correction requires $v^3 o$ storage and scales as $v^4 o^3$. A naive (T) algorithm is trivial to implement but an algorithm with a small memory and scalable I/O requires thought.

There is a CCSD(T) method in nearly every quantum chemistry package. The state-of-the-art is ACES [19]; its implementation can handle very large systems on a large cluster. NWChem [20] implementation is slower but is similarly scalable. The MOLPRO [21] algorithm has $o^2 v^2$ memory requirement, which limits its utility, but it is perhaps the fastest there is for smaller calculations. The GAMESS [22] implementation runs in parallel but is similarly limited by $o^2 v^2$ memory requirement.

Only two implementations, the NWChem and ACES, are able to handle computations of the order of thousand basis functions, but both require access to fairly large machines to do so. See for example ACES Parallel Coupled Cluster benchmarks [23].

2 Design of a Scalable and Efficient Algorithm

Previously, we reported an MP2 energy algorithm, which has small memory footprint, good performance, flexible storage implementation, and is able to run on workstations and clusters equally well. In the same spirit, a coupled cluster algorithm can be designed, such that it is efficient, has small memory footprint, is able to utilize filesystem and memory for storage, and as a result can run on machines with very different capabilities.

As far as the coupled cluster algorithms (and other many-body methods) are concerned, it is the memory that is most likely to limit the application of the algorithm. Memory is a limited resource, unlike the time. Furthermore, calculations can be sped up by throwing more computational hardware at a problem, whereas the amount of physical memory per node can not be increased by adding another node.

Some very large arrays can be (and need to be) distributed across the nodes (distributed memory) or stored on the filesystem. The disks are cheap and offer terabytes of storage, but filesystem I/O can be very slow if not done right. Nevertheless, a considerable amount of memory must be present to carry out local calculations.

What are the memory limitations of the current hardware? On a *typical* workstation or a cluster node, there is anywhere between 1GB and 8GB per core, with 2GB of RAM (or less) *much* more common than, say, 4GB per *core*. Per entire *node*, the amount of memory can vary from 2GB up to 64GB and more, with 8-16GB fairly common. That number will increase in the future, but at much slower rate than the increase in computational power.

To draw a connection between memory and the dimensions present in the CC calculations, several arrays, corresponding to 100 occupied orbitals and 1000 and 2000 basis functions, are listed in Table 2. It should be kept in mind that the sizes listed are not for the entire calculation, but for one of the several arrays needed. Some of the arrays can be shared, but some must be allocated per thread/core.

it should be clear that storing o^2n^2 quantity per node (let alone per core) is too expensive: a node with 80GB of RAM is rare and one with 320GB even more so. The same goes for the quartic arrays other than o^4 and arrays involving n^2 factor: storing for example several 3GB arrays would preclude most systems from being able to handle more than a thousand basis functions. In the end we are left with restricting memory requirements to o^2n (or smaller) arrays, which size only increases linearly with basis set. Trying to limit memory further than o^2n, to say on, will come at the very high cost of increased I/O.

Some arrays, notably n^4, are too great to store even in the secondary storage. The terms involving that array need to be evaluated directly, i.e.on the fly, at the modest cost of recomputing atomic integrals, cf. [24]. However, to push the ability of the algorithm beyond a thousand basis function, on^3 storage also needs to be eliminated in the CCSD algorithm. To ensure that I/O overhead is low even on filesystems, transfers to and from secondary storage must be contigous and in large chunks. There are three basic remote operations: `put, get, accumulate`. The last one cannot be implemented efficiently via the filesystem I/O and the algorithm must not rely on it.

Finally, to achieve computational efficiency, all the expensive tensor contractions need to be carried out using `dgemm` and tensor permutations must not exceed two adjecent indices, e.g. $A(j,i,k) = A(i,j,k)$ is ok, but $A(k,j,i) = A(i,j,k)$ is not. The work distribution between the nodes must be over the virtual index rather than

Table 1: Array Sizes for o=100

Array	Size (GB), n=1000	Size (GB), n=2000
o^4	0.8	0.8
o^2n	0.8	1.6
on^2	0.8	3.2
o^3n	8.0	16.0
n^3	8.0	64.0
o^2n^2	80.0	320.0
on^3	800.0	6400.0
n^4	8000.0	128000.0

much smaller occupied index, to ensure that the algorithm can scale into hundreds of nodes. The work within the node can be parallelized using threads. This multi-level parallelization guarantees that the algorithm will scale into thousands of cores.

The order in which algorithm points are listed are in the order in which the algorithm was thought of. Essentially, the primary focus is on memory, then secondary storage and I/O, and only then on the computational aspect. It should be noted that despite that way of thinking the performance does not suffer, as will be illustrated with benchmarks.

3 Implementation

This section is broken down into three: the direct CCSD terms, the non-direct CCSD terms, and triples correction. The CCSD is by far the most complex due to number of terms.

Before proceeding to the respective sections, a word must be said about optimizing I/O via loop blocking. Consider the Algorithm 2, where B is a blocking factor. If $B = 1$, then it is just a regular loop: the innermost (most expensive) load operation is executed N^3 times, the total I/O overhead is M^2N^3, and the local buffer size is M^2. If B is greater than 1, the innermost load operation is called $(N/B)^3$ times, the I/O overhead is $M^2B(N/B)^3 = M^2N^3/B^2$, and the local buffer size is M^2B. So, at the cost of increasing local buffer, the I/O overhead can be reduced by a factor of B^2. In general, loop blocking decreases I/O by $B^{(L-1)}$ where L is the loop depth.

The loop blocking will be used where I/O might pose a problem. Since blocking also requires increase in memory overhead, the blocking factor can be determined by setting runtime memory limit.

3.1 Direct Terms

As mentioned already, the v_{cd}^{ab} has to be evaluated directly due to storage constraints. The same approach can be extended to evaluate terms with v_{bc}^{ia} directly as well at little additional cost.

The v_{cd}^{ab} is contracted with $T_{1\,ij}^{2\,cd} = t_i^c t_j^d$ and $T_{2\,ij}^{cd} = t_{ij}^{cd}$ amplitudes,

```
for i = 0:N,B { // iterate to N in steps of B
  for j = 0:N,B {
    for k = 0:N,B {
      // the innermost load operation
      buffer(M,M,B) = load A(M,M,k:k+B)
      ...
    }
  }
}
```

Table 2: Loop Blocking

$$VT_{2\,ij}^{ab} = t_{ij}^{cd} C_d^s C_c^q V_{qs}^{pr} C_r^b C_p^a$$

$$VT_{1\,ij}^{2\,ab} = t_i^c t_j^d C_d^s C_c^q V_{qs}^{pr} C_r^b C_p^a$$

Half-transforming the amplitudes to AO basis and factoring out half-contracted terms yields expressions in terms of half-transformed intermediates U,

$$U_{2\,ij}^{qs} = (t_{ij}^{cd} C_d^s C_c^q) V_{qs}^{pr}$$

$$U_{1\,ij}^{2\,qs} = (t_i^c C_c^q)(t_j^d C_d^s) V_{qs}^{pr}$$

$$VT_{2\,ij}^{ab} = U_{2\,ij}^{qs} C_s^b C_q^a$$

$$VT_{1\,ij}^{2\,ab} = U_{1\,ij}^{2\,qs} C_s^b C_q^a$$

All similar VT terms can be obtained from U at virtually no cost by having the last two AO indices transformed to occupied and virtual index. For example, the v_{bc}^{ia} terms in Eq. 1 is just

$$v_{ef}^{ma}(2t_{mi}^{ef} - t_{im}^{ef}) = 2U_{mi}^{qs} C_q^m C_s^a - U_{im}^{qs} C_q^m C_s^a$$

The v_{bc}^{ia} also enters VT_1 diagrams,

$$VT_{1\,ab}^{ij} = t_i^c C_j^s C_c^q V_{qs}^{pr} C_r^b C_p^a$$

$$VT_{1\,jb}^{ia} = t_i^c C_d^s C_j^q V_{qs}^{pr} C_r^b C_p^a$$

and two more intermediates are needed,

$$U_{1\,qs}^{ij} = (t_a^i C_p^a) C_r^j V_{qs}^{pr}$$

$$U_{1\,js}^{ir} = (t_a^i C_p^a) C_j^q V_{qs}^{pr}$$

which can then be transformed into appropriate VT_1 diagrams.

```
for S in Shells {
  for Q <= S {
    for R in Shells {
      for P in Shells {
        // skip insignificant ints
        if (!screen(P,Q,R,S)) continue;
        G(P,R,Q,S) = eri(P,Q,R,S);
      }
      for r in R {
        U1(i,j,q,s) = ...
        U12(i,j,q,s) = ...
        load t(o,o,n,r)
        U2(i,j,q,s) += t(i,j,p,r)*G(p,r,q,s)
      }
    }
    store U1(i,j,Q,S), U1(j,i,S,Q)
    store U12(i,j,Q,S), U12(j,i,S,Q)
    store U2(i,j,Q,S), U2(j,i,S,Q)
  }
}
```

Table 3: Direct CCSD intermediates

Now, if all four U intermediates are available, neither v_{cd}^{ab} nor v_{bc}^{ia} need be stored for the CCSD iterations, replaced with much smaller $4o^2n^2$ storage.

Half-transformed T_2 amplitudes also give a way to devise a direct contraction algorithm with very little memory requirement. Recall that the contraction is in AO basis, and thus atomic indices can be contracted without having to construct V_{qs}^{ab} which would require all p, r indices and thus N^2M^2 memory, where M is the size of the largest shell. The algorithm in Listing 3 only needs NM^3 memory.

The important points of the algorithm 3 are:

- The integral symmetry is exploited to halve integral calculation *and* transformations.
- The loop over Q, S can be distributed over nodes.
- The loop over R can be parallelized over threads. In this case, the U storage can be shared, provided the updates to shared memory are synchronized.
- The innermost $t2$ loads can be reduced by bloking Q, S loops.
- Per thread storage is NM^3, which is 16MB for a basis set of size 2000 with f shells ($M = 10$). The local U storage is likewise small, only for 8MB $o = 100$. This tiny memory footprint allows for a very large Q, S blocking factor and consequently

the I/O can be drastically reduced.

A careful reader may have noticed that both UT_1 terms can not be evaluated simultaneously using the above algorithm, as they corresponds to two different integrals, $\langle pq|rs\rangle$ and $\langle pr|qs\rangle$. However, one of them can be easily evaluated by applying the algorithm a second time to compute a single UT_1 term at a very modest on^4 cost.

3.2 CCSD

Because the singles amplitudes storage is negligible, on, the singles is easy to implement and parallelize. By making a virtual index outermost, the local memory is guaranteed not to exceed o^2n since all the diagrams with three and four virtual indices have already been evaluated above.

The doubles amplitudes calculation requires the most effort to implement, primarily due to the number of contractions and the terms which require significant I/O. Keep in mind that all v_{cd}^{ab} and v_{bc}^{ia} terms have been evaluated, but so have been many similar VT terms.

The first step towards deriving a scalable algorithm for Dt_{ab}^{ij} is to fix the outermost loop at b index, which can be evaluated across nodes independently. For each b iteration a o^2n Dt block is evaluated and stored.

The quantities with b index are loaded once, guaranteed not to exceed o^2n size. The tensors without b index imply that the tensor is needed in its entirety for each index. To ensure that no v or t memory exceeds o^2n, those tensors must be loaded into memory o^2n tile at a time for each b index inside a loop over a dummy virtual index, lets call it u. This increases the I/O cost to o^2n^3, which is still below the o^3n^3 computational cost.

There are three tensors which must be contracted fully for a given b index: $v_{ia}^{jb}, v_{ij}^{ab}, t_{ij}^{ab}$. The loop corresponding to v_{ia}^{jb} can be elimiated right away, it is only needed in its entirity to evaluate $I_{ie}^{ma}t_{mj}^{eb}$ in Eq. 2. Since, this term appears inside the symmetrizer P,

$$P(v_{je}^{mb}t_{mi}^{ea}) = P(v_{ie}^{ma}t_{mj}^{eb})$$

I_{ie}^{ma} can be replaced by an equivalent I_{je}^{mb}. This leads to Algorithm 4.

The important points about the Algorithm 4 are:

- The loop over b index is easy to make parallel.
- The local memory is on the order $4o^2n$ plus o^2n per innermost v'/t' memory.
- The b loop can be easily blocked to reduce the I/O by a blocking factor B at the expense of increasing memory by a factor of B.
- Since than memory footprint is low, B can be fairly large. E.g. for O=100, V=2000, B=4 and B=8, the required memory is 2.6G and 5.2G per *node* respectively.
- The operations outside the u loop can be parallelized inside the node by using threaded math library.
- The operations inside the u loop can be explicitly parallelized inside the node via threads, with the added benefit of overlapping I/O and computations.

```
for b in v {
  Dt(i,j,a) = 0

  load t(o,o,v,b)
  load V(o,o,v,b)
  load V(o,v,o,b)
  load V(o,o,o,b)

  Dt += Vt

  // terms with t
  for u in v {
    load t'(o,o,v,u)
    // evaluate terms with t'
    Dt += Vt'
  }

  // terms with v
  for u in v {
    load v'(o,o,v,u)
    // evaluate terms with v'
    Dt += V't
  }

  store Dt(o,o,v,b)
}
```

Table 4: CCSD Algorithm

3.3 (T)

The (T) correction, Eq. 5, only involves t_{ab}^{ij}, v_{ka}^{ij}, v_{ab}^{ij}, and v_{bc}^{ia}. The unused CCSD arrays previously allocated can be freed to make space for v_{bc}^{ia}. Since v_{bc}^{ia} was never constructed, another integral transformation needs to be carried out at a small on^4 cost.

The original (T) correction equations were given in a way that requires keeping occupied index fixed and permuting the virtual index, in other words the local memory for t_{abc}^{ijk} would have been v^3. Since triple amplitude are symmetric with respect to exchange of index "columns",

$$t_{abc}^{ijk} = t_{bac}^{jik} = t_{acb}^{ikj} = ...$$

all terms with t_{bac}^{jik} can be written with virtual index fixed, e.g. $t_{bac}^{ijk} = t_{abc}^{jik}$, $t_{cab}^{ijk} = t_{abc}^{jki}$, etc.

Now the T_3 amplitudes can be implemented as a series of 12 `dgemms` and 6 index permutations, Listing 5. The important points about the algorithm are:

- The symmetry in a, b, c indices is utilized.
- The loop over a, b, c indices is easily parallelizable.
- Only the loads with a index are innermost
- The loops can be easily blocked to reduce the I/O by a factor of B^2 where B is the blocking factor.
- The local storage required is $3o^2vB + 3o^3B + 6ovB^2 + o^3B^3$
- If $B > 1$, the actual `dgemms` are carried out inside another B^3 loop, which can be parallelized *within* a node by the means of threads.
- Since the memory footprint is low, blocking factor can be large. E.g. for O=100, V=1000, B=4 and B=8, the required memory is 1.6G and 6.4G per *node* respectively.

3.4 The overall picture.

The algorithm is implemented entirely in C++, as a part of stand-alone library which includes previous ERI, Fock, and MP2 methods [25, **?**]. The library requires only minimal input from the host program and can be connected to a variety of packages.

The storage is implemented using Global Arrays (GA) [26] for distributed memory and HDF5 [27] for file storage. GAMESS's own distributed memory interface (DDI) [28] currently has no support for arrays of more than 2 dimensions. But with a small addition, the DDI calls can be translated directly into GA equivalents, so that the GAMESS can run via GA without modifications while at the same time providing 3-d and 4-d array functionality via direct calls to GA.

The arrays are first allocated in faster GA memory until the limit is reached, and then on the filesystem. The arrays responsible for the most I/O need to be allocated first to ensure they reside in distributed memory. The implementation is a separate library, linked to

Put together, the algorithm looks like this:

```
for c in V {
  for b in c {
    for a in b {

      load t(o,o,a,b)
      load t(o,o,a,c)
      load t(o,o,b,c)

      load v(o,o,o,a)
      load v(o,o,o,b)
      load v(o,o,o,c)

      load v(o,o,v,a)
      load v(o,o,v,b)
      load v(o,o,v,c)

      load v(o,v,b,c)
      load v(o,v,c,b)
      load v(o,v,a,c)
      load v(o,v,c,a)
      load v(o,v,a,b)
      load v(o,v,b,a)

      // t(i,j,e,a)*V(e,k,b,c) corresponds to dgemm(t(ij,e), V(e,k)), etc
      t(i,j,k) = t(i,j,e,a) V(e,k,b,c) - t(i,m,a,b) V(j,k,m,c)
      t(i,k,j) = t(i,k,e,a) V(e,j,c,b) - t(i,m,a,c) V(k,j,m,b)
      t(k,i,j) = t(k,i,e,c) V(e,j,a,b) - t(k,m,c,a) V(i,j,m,b)
      t(k,j,i) = t(k,j,e,c) V(e,i,b,a) - t(k,m,c,b) V(j,i,m,a)
      t(j,k,i) = t(j,k,e,b) V(e,i,a,c) - t(j,m,b,c) V(k,i,m,a)
      t(j,i,k) = t(j,i,e,b) V(e,k,c,a) - t(j,m,b,a) V(i,k,m,c)
      ...
    }
  }
}
```

Table 5: (T) Algorithm

Table 6: Single Node Performance

Input	#AO/Occ	CCSD	(T)	(T) Mem/Disk	(T) I/O
C4N3H5/aug-ccPVTZ	565/21	42m	8h	2.1/19.5 GB	13m
C8H10N4O2/aug-ccPVDZ	440/37	50m	17h	5.5/17.0 GB	13m
SiH4B2H6/aug-ccPVQZ	875/16	141m	18h	3.4/53.4 GB	49m
C8H10N4O2/ccPVTZ	640/37	180m	64h	12.2/49.0 GB	42m

- The CCSD arrays are allocated, with t and v_{ij}^{ab} first to ensure they are in the fast storage. Overall, t, v_{ij}^{ab}, v_{ij}^{ka}, v_{ia}^{jb}, v_{ij}^{kl}, Dt, storage is needed.
- The allocated arrays are evaluated using the regular 4-index transformation.
- The initial $T2$ amplitudes are taken to be the MP2 amplitudes, v_{ij}^{ab}/D_{ij}^{ab} and the $T1$ amplitudes are set to zero.
- The intermediate U storage is allocated.
- The CCSD equations are repeated until an acceptable threshold is reached, either the energy difference or the amplitude difference.
- The CCSD step is optionally accelerated using DIIS [18].
- Once converged, all but the first three arrays are freed and v_{ia}^{bc} array is allocated and and evaluated.
- The non-iterative (T) method runs.

4 Performance

To assess performance and applicability of the algorithm, the three case scenarios are presented: single node performance, departmental cluster performance, and high-end cluster performance. The inputs are selected to reflect a range of basis functions and occupied orbitals.

The departamental cluster, Exalted, is composed of nodes connected by InfiniBand Ethernet. Each node has one Intel X5550 2.66GHz 6-core processor, 24GB of RAM, two local disk drives, and an NVIDIA Fermi C2050 GPU card.

First, the capability of the algorithm to run on a single node and to use filesystem in case not enough memory is available to store all data. The results four different inputs are in table 4. As can be seen, even on a single node fairly large jobs can still run in a reasonable timeframe. Despite falling back to disk in all the cases, across the board the I/O time as a fraction of total time is very small, below 5%.

The cluster performance is assessed on the basis of the time larger jobs take to run, Table 4, and the scalability of a medium-size job, Table 4. First, all of the inputs used for single node benchmarking can run in under a day. Secondly, a large CCSD Tamoxifen calculation, C26H29NO, can run on this relatively small Exalted cluster, three hours per iteration.

As can be expected, the (T) algorithm scales well, Table 4, since it is very easy to parallelize to a large number of nodes. However, the scalability of the CCSD algorithm

Table 7: Cluster Performance

Input	#AO/Occ	# cores	CCSD	(T)
C4N3H5/aug-ccPVTZ	565/21	4	12	117
SiH4B2H6/aug-ccPVQZ	875/16	8	20	133
C8H10N4O2/ccPVTZ	640/37	8	26	482
C26H29NO/aug-ccPVQZ	961/71	16	211	N/A

Table 8: Cluster Scaling, C8H10N4O2/ccPVTZ

Cores/Nodes	CCSD	(T)
24/4	28	971
48/8	15	482
96/16	11	240

is not perfect. This is especially noticable when running on a large Cray system, Table 4. Nevertheless, the longer Tamoxfine calculation scales reasonably well to 1024 cores.

Each XE6 has to chips, 16 cores each. The benchmarks in Table 4 were obtained running 32 threads over the entire node spawned from a single MPI process. the better option, especially in the case of (T) is to run one MPI process per *core* rather than per *node*. In the former case, the threads do not need to communicate over the slower bridge connecting two chips, Table **??**. Generally, there is a large penalty for sharing data across the *chips*, which must be avoided by having flexible approach to launch jobs.

4.1 GPU

As can be expected, the direct terms account for the most time in CCSD iterations. The favorable fact about implementation is that most of that work is concentrated in continuous application of just one *dgemm* operation. Adding a graphical processor (GPU) *dgemm* to handle matrix multiplication, while keeping integral evaluation on the host, is fairly easy. However, in multithreaded environment, several threads must be assigned to a GPU device to avoid work imbalance.

Augmented with GPU BLAS, via CUBLAS [29], the CCSD calculations get a no-

Table 9: Cray XE6 Performance

# cores	256	512	1024
SiH4B2H6 (T)/aug-ccPVQZ	130	76	42
C8H10N4O2 CCSD/ccPVTZ	15	9	6.5
C26H29NO CCSD/aug-ccPVQZ	253	134	76

Table 10: Cray XE6 Intra-Node Configuration, SiH4B2H6 (T)/aug-ccPVQZ

# cores	32x1 Threads/MPI	16x2 Threads/MPI
256	130	83
512	76	49
1024	42	27

Table 11: Gpu CCSD performance, all times are in minutes per iteration

Input	C8H10N4O2/ccPVTZ	SiH4B2H6/aug-ccPVQZ	C4N3H5/aug-ccPVTZ
Direct	124	131	36
Direct+GPU	53	65	26
CCSD	163	142	42
CCSD+GPU	115	75	33
Speed-up	1.4x	1.9X	1.3X

ticeable speed up, Table 4.1, if the direct term dominates the entire iteration (this is the case if number of occupied orbitals is very small relative to basis set). If the number of occupied orbitals is a relatively high, the direct term accounts for a smaller fraction of the total iteration time, and consequently GPU benefit is less noticeable overall.

5 Conclusions

The algorithm presented in this paper is able to handle fairly large jobs on a single node, a small cluster, and high-end Cray system. The algorithm has reduced memory footprint and is able to optionally use the filesystem if the data exceeds distributed memory storage. The algorithm can also optionally use GPUs to speed up certain CCSD computations. When running on the multi-core node with multiple processor packages (chips), algorithm benefits from limiting thread communication to within a chip.

The algorithm is implemented entirely in C++, as a part of stand-alone library which includes previous ERI, Fock, and MP2 methods. [**?**].

References

[1] JA Pople, R. Seeger, and R. Krishnan. Variational configuration interaction methods and comparison with perturbation theory. *International Journal of Quantum Chemistry*, 12(S11):149–163, 1977.

[2] J.A. Pople, M. Head-Gordon, and K. Raghavachari. Quadratic configuration interaction. a general technique for determining electron correlation energies. *The Journal of chemical physics*, 87(10):5968–5975, 1987.

[3] J.A. Pople, P.M.W. Gill, and B.G. Johnson. Kohnsham density-functional theory within a finite basis set. *Chemical physics letters*, 199(6):557–560, 1992.

[4] K. Raghavachari, G.W. Trucks, J.A. Pople, and M. Head-Gordon. A fifth-order perturbation comparison of electron correlation theories. *Chemical Physics Letters*, 157(6):479–483, 1989.

[5] F. Coester and H. Kümmel. Short-range correlations in nuclear wave functions. *Nuclear Physics*, 17:477–485, 1960.

[6] J. Čížek. On the correlation problem in atomic and molecular systems. calculation of wavefunction components in ursell-type expansion using quantum-field theoretical methods. *The Journal of Chemical Physics*, 45(11):4256–4266, 1966.

[7] J. Paldus, J. Čížek, and I. Shavitt. Correlation problems in atomic and molecular systems. iv. extended coupled-pair many-electron theory and its application to the bh_ {3} molecule. *Physical Review A*, 5(1):50, 1972.

[8] D. Mukherjee, R.K. Moitra, and A. Mukhopadhyay. Applications of a non-perturbative many-body formalism to general open-shell atomic and molecular problems: calculation of the ground and the lowest π-π^* singlet and triplet energies and the first ionization potential of trans-butadiene. *Molecular Physics*, 33(4):955–969, 1977.

[9] I. Lindgren. A coupled-cluster approach to the many-body perturbation theory for open-shell systems. *International Journal of Quantum Chemistry*, 14(S12):33–58, 1978.

[10] G.E. Scuseria, C.L. Janssen, and H.F. Schaefer. An efficient reformulation of the closed-shell coupled cluster single and double excitation (ccsd) equations. *The Journal of Chemical Physics*, 89(12):7382–7387, 1988.

[11] P. Piecuch, S.A. Kucharski, and R.J. Bartlett. Coupled-cluster methods with internal and semi-internal triply and quadruply excited clusters: Ccsdt and ccsdtq approaches. *The Journal of chemical physics*, 110:6103, 1999.

[12] G.D. Purvis III and R.J. Bartlett. A full coupled-cluster singles and doubles model: The inclusion of disconnected triples. *The Journal of Chemical Physics*, 76:1910, 1982.

[13] A. Szabo and N.S. Ostlund. *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory*. Dover Books on Chemistry Series. Dover Publications, 1996.

[14] T. Janowski, A.R. Ford, and P. Pulay. Parallel calculation of coupled cluster singles and doubles wave functions using array files. *Journal of Chemical Theory and Computation*, 3(4):1368–1377, 2007.

[15] D. Yarkony. *Modern Electronic Structure Theory*. Number pt. 2 in Advanced Series in Physical Chemistry. World Scientific, 1995.

[16] I. Shavitt and R.J. Bartlett. *Many-Body Methods in Chemistry and Physics: MBPT and Coupled-Cluster Theory*. Cambridge Molecular Science. Cambridge University Press, 2009.

[17] P. Piecuch, S.A. Kucharski, K. Kowalski, and M. Musiał. Efficient computer implementation of the renormalized coupled-cluster methods: The r-ccsd [t], r-ccsd

(t), cr-ccsd [t], and cr-ccsd (t) approaches. *Computer Physics Communications*, 149(2):71–96, 2002.

[18] G.E. Scuseria, T.J. Lee, and H.F. Schaefer. Accelerating the convergence of the coupled-cluster approach: The use of the diis method. *Chemical physics letters*, 130(3):236–239, 1986.

[19] V. Lotrich, N. Flocke, M. Ponton, AD Yau, A. Perera, E. Deumens, and RJ Bartlett. Parallel implementation of electronic structure energy, gradient, and hessian calculations. *The Journal of chemical physics*, 128:194104, 2008.

[20] DE Bernholdt, E. Apra, HA Früchtl, MF Guest, RJ Harrison, RA Kendall, RA Kutteh, X. Long, JB Nicholas, JA Nichols, et al. Parallel computational chemistry made easier: The development of nwchem. *International Journal of Quantum Chemistry*, 56(S29):475–483, 1995.

[21] H.J. Werner, PJ Knowles, R. Lindh, FR Manby, M. Schütz, P. Celani, T. Korona, G. Rauhut, RD Amos, A. Bernhardsson, et al. Molpro, version 2006.1, a package of ab initio programs. 2006.

[22] M. S. Gordon and M. W. Schmidt. *Advances in electronic structure theory: GAMESS a decade later*, pages 1167–1189. Elsevier, Amsterdam, 2005.

[23] Pcc benchmarks. http://www.qtp.ufl.edu/PCCworkshop/PCCbenchmarks.html.

[24] R.M. Olson, J.L. Bentz, R.A. Kendall, M.W. Schmidt, and M.S. Gordon. A novel approach to parallel coupled cluster calculations: Combining distributed and shared memory techniques for modern cluster based systems. *Journal of Chemical Theory and Computation*, 3(4):1312–1328, 2007.

[25] A. Asadchev, V. Allada, J. Felder, B.M. Bode, M.S. Gordon, and T.L. Windus. Uncontracted rys quadrature implementation of up to g functions on graphical processing units. *Journal of Chemical Theory and Computation*, 6(3):696–704, 2010.

[26] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–231, 2006.

[27] The HDF Group. Hierarchical data format, Version 5. http://www.hdfgroup.org/HDF5.

[28] R.M. Olson, M.W. Schmidt, M.S. Gordon, and A.P. Rendell. Enabling the efficient use of smp clusters: the gamess/ddi model. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 41. ACM, 2003.

[29] R. Nath, S. Tomov, and J. Dongarra. Accelerating gpu kernels for dense linear algebra. *High Performance Computing for Computational Science–VECPAR 2010*, pages 83–92, 2011.

References

[1] JA Pople, R. Seeger, and R. Krishnan. Variational configuration interaction methods and comparison with perturbation theory. *International Journal of Quantum Chemistry*, 12(S11):149–163, 1977.

[2] J.A. Pople, M. Head-Gordon, and K. Raghavachari. Quadratic configuration interaction. a general technique for determining electron correlation energies. *The Journal of chemical physics*, 87(10):5968–5975, 1987.

[3] J.A. Pople, P.M.W. Gill, and B.G. Johnson. Kohnsham density-functional theory within a finite basis set. *Chemical physics letters*, 199(6):557–560, 1992.

[4] K. Raghavachari, G.W. Trucks, J.A. Pople, and M. Head-Gordon. A fifth-order perturbation comparison of electron correlation theories. *Chemical Physics Letters*, 157(6):479–483, 1989.

[5] F. Coester and H. Kümmel. Short-range correlations in nuclear wave functions. *Nuclear Physics*, 17:477–485, 1960.

[6] J. Čížek. On the correlation problem in atomic and molecular systems. calculation of wavefunction components in ursell-type expansion using quantum-field theoretical methods. *The Journal of Chemical Physics*, 45(11):4256–4266, 1966.

[7] J. Paldus, J. Čížek, and I. Shavitt. Correlation problems in atomic and molecular systems. iv. extended coupled-pair many-electron theory and its application to the bh_ {3} molecule. *Physical Review A*, 5(1):50, 1972.

[8] D. Mukherjee, R.K. Moitra, and A. Mukhopadhyay. Applications of a non-perturbative many-body formalism to general open-shell atomic and molecular problems: calculation of the ground and the lowest π-π^* singlet and triplet energies and the first ionization potential of trans-butadiene. *Molecular Physics*, 33(4):955–969, 1977.

[9] I. Lindgren. A coupled-cluster approach to the many-body perturbation theory for open-shell systems. *International Journal of Quantum Chemistry*, 14(S12):33–58, 1978.

[10] G.E. Scuseria, C.L. Janssen, and H.F. Schaefer. An efficient reformulation of the closed-shell coupled cluster single and double excitation (ccsd) equations. *The Journal of Chemical Physics*, 89(12):7382–7387, 1988.

[11] P. Piecuch, S.A. Kucharski, and R.J. Bartlett. Coupled-cluster methods with internal and semi-internal triply and quadruply excited clusters: Ccsdt and ccsdtq approaches. *The Journal of chemical physics*, 110:6103, 1999.

[12] G.D. Purvis III and R.J. Bartlett. A full coupled-cluster singles and doubles model: The inclusion of disconnected triples. *The Journal of Chemical Physics*, 76:1910, 1982.

[13] A. Szabo and N.S. Ostlund. *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory*. Dover Books on Chemistry Series. Dover Publications, 1996.

[14] T. Janowski, A.R. Ford, and P. Pulay. Parallel calculation of coupled cluster singles and doubles wave functions using array files. *Journal of Chemical Theory and Computation*, 3(4):1368–1377, 2007.

[15] D. Yarkony. *Modern Electronic Structure Theory*. Number pt. 2 in Advanced Series in Physical Chemistry. World Scientific, 1995.

[16] I. Shavitt and R.J. Bartlett. *Many-Body Methods in Chemistry and Physics: MBPT and Coupled-Cluster Theory*. Cambridge Molecular Science. Cambridge University Press, 2009.

[17] P. Piecuch, S.A. Kucharski, K. Kowalski, and M. Musiał. Efficient computer implementation of the renormalized coupled-cluster methods: The r-ccsd [t], r-ccsd (t), cr-ccsd [t], and cr-ccsd (t) approaches. *Computer Physics Communications*, 149(2):71–96, 2002.

[18] G.E. Scuseria, T.J. Lee, and H.F. Schaefer. Accelerating the convergence of the coupled-cluster approach: The use of the diis method. *Chemical physics letters*, 130(3):236–239, 1986.

[19] V. Lotrich, N. Flocke, M. Ponton, AD Yau, A. Perera, E. Deumens, and RJ Bartlett. Parallel implementation of electronic structure energy, gradient, and hessian calculations. *The Journal of chemical physics*, 128:194104, 2008.

[20] DE Bernholdt, E. Apra, HA Früchtl, MF Guest, RJ Harrison, RA Kendall, RA Kutteh, X. Long, JB Nicholas, JA Nichols, et al. Parallel computational chemistry made easier: The development of nwchem. *International Journal of Quantum Chemistry*, 56(S29):475–483, 1995.

[21] H.J. Werner, PJ Knowles, R. Lindh, FR Manby, M. Schütz, P. Celani, T. Korona, G. Rauhut, RD Amos, A. Bernhardsson, et al. Molpro, version 2006.1, a package of ab initio programs. 2006.

[22] M. S. Gordon and M. W. Schmidt. *Advances in electronic structure theory: GAMESS a decade later*, pages 1167–1189. Elsevier, Amsterdam, 2005.

[23] Pcc benchmarks. http://www.qtp.ufl.edu/PCCworkshop/PCCbenchmarks.html.

[24] R.M. Olson, J.L. Bentz, R.A. Kendall, M.W. Schmidt, and M.S. Gordon. A novel approach to parallel coupled cluster calculations: Combining distributed and shared memory techniques for modern cluster based systems. *Journal of Chemical Theory and Computation*, 3(4):1312–1328, 2007.

[25] A. Asadchev, V. Allada, J. Felder, B.M. Bode, M.S. Gordon, and T.L. Windus. Uncontracted rys quadrature implementation of up to g functions on graphical processing units. *Journal of Chemical Theory and Computation*, 6(3):696–704, 2010.

[26] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–231, 2006.

[27] The HDF Group. Hierarchical data format, Version 5. http://www.hdfgroup.org/HDF5.

[28] R.M. Olson, M.W. Schmidt, M.S. Gordon, and A.P. Rendell. Enabling the efficient use of smp clusters: the gamess/ddi model. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 41. ACM, 2003.

[29] R. Nath, S. Tomov, and J. Dongarra. Accelerating gpu kernels for dense linear algebra. *High Performance Computing for Computational Science–VECPAR 2010*, pages 83–92, 2011.