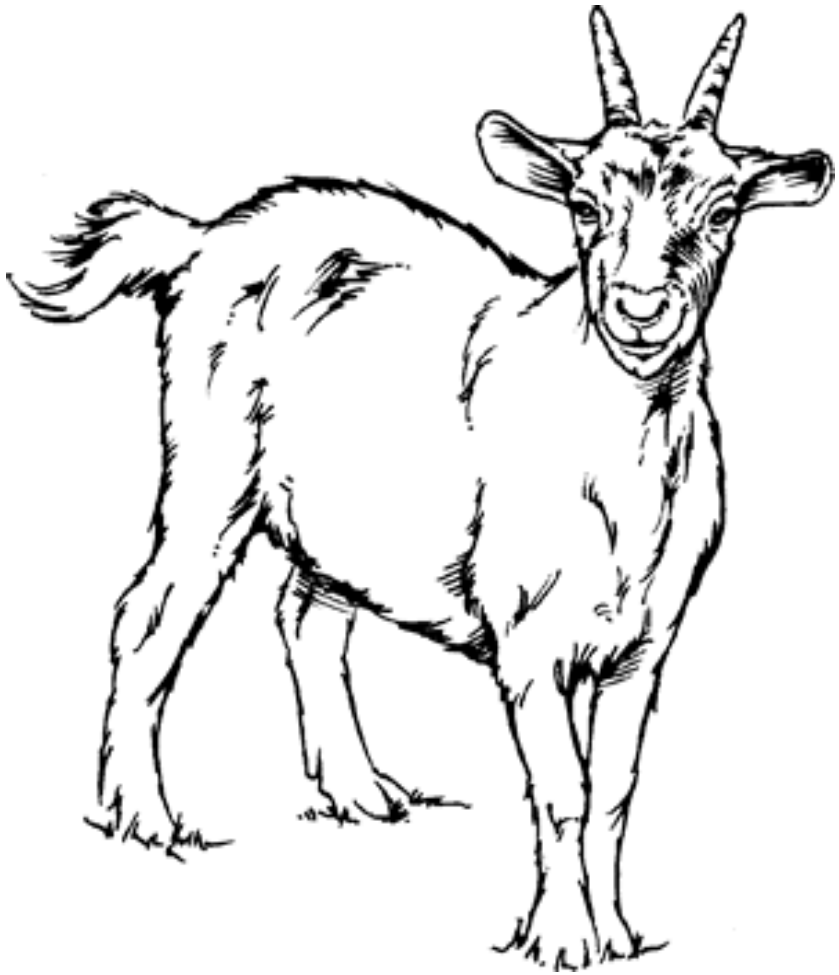
Valeev Group Meeting

Andrey Asadchev

VT.edu

September 27, 2013

Picture of a goat!!!



Configuration Interaction on a conceptual level

- Assume that we have N_α alpha electrons, N_β beta electrons, and K orbitals.
- Then there are $\binom{K}{N_\alpha}$ alpha and $\binom{K}{N_\beta}$ arrangements.
- Lets choose represent empty orbitals with 0s and occupied orbitals with 1s.
- Then electron arrangements, called strings, can be represented as a *bit* string
- For example $K = 4, N = 2$ configuration will generate $1100, 1010, 1001, 0110, 0101, 0011$
- The number of strings grows quickly, $K = 32, N = 16$ will generate over 600 million strings.

Configuration Interaction on a conceptual level

- A determinant is an arrangement of alpha and beta electrons, which is really a pair of alpha and beta strings
- Strings with all electrons occupying the lowest orbitals, ie the reference string, can be (arbitrarily) chosen to be 11..00 and any other strings are said to be excited.
- Similarly, the determinants are either reference or (singly, doubly, ...) excited.
- CI expression $|\Psi\rangle = c_0\langle\Phi_0| + \sum c_i^a\langle\Phi_i^a| + ..$ implemented in terms of determinants is called determinant CI.
- There are other types of CI but the determinant CI has a huge computational advantage ...

Determinant CI

- There are $\binom{K}{N_\alpha}$ alpha and $\binom{K}{N_\beta}$ beta strings.
- In Full CI there are then $\binom{K}{N_\alpha}\binom{K}{N_\alpha}$ determinants.
- But the entire Hamiltonian is square the number of determinants!!!
- Only for smallest of spaces can the Hamiltonian be evaluated in full and diagonalized.
- For anything larger, the CI roots (eigenvalues) and coefficient vectors (CI vectors) need to be evaluated iteratively.
- To do that fast, various Hamiltonian elements, eg $\langle I_\alpha I_\beta | H | J_\alpha J_\beta \rangle$, need to be evaluated on the fly, fast.
- With determinant CI evaluation of those element can be fast.

The three CI parts

- The three types of CI excitations are alpha-alpha, beta-beta, and simultaneous alpha-beta.
- Those are commonly referred to as $\sigma_1, \sigma_2, \sigma_3$.

$$\sigma_1(a,b) = \sum_{b'}\sum_{ij} \langle b|E_{ij}^{\beta}|b'\rangle h'_{ij} C(a,b') + \sum_{b'}\sum_{ijkl} \langle b|E_{ij}^{\beta}|b'\rangle \langle b|E_{kl}^{\beta}|b'\rangle (ij|kl) C(a,b')$$

$$\sigma_2(a,b) = \sum_{a'}\sum_{ij} \langle a|E_{ij}^{\alpha}|a'\rangle h'_{ij} C(a',b) + \sum_{a'}\sum_{ijkl} \langle a|E_{ij}^{\alpha}|a'\rangle \langle a|E_{kl}^{\alpha}|a'\rangle (ij|kl) C(a',b)$$

$$\sigma_3(a,b) = \sum_{a'b'}\sum_{ijkl} \langle a|E_{ij}^{\alpha}|a'\rangle \langle b|E_{kl}^{\beta}|b'\rangle (ij|kl) C(a',b')$$

The three CI parts

- In a nutshell, each string has a set of allowed excitation and with it associated a phase, string position in the list, and the integral indices.
- I'll use notation $E_{ij}^{aa'}$ to signify allowed excitation from a to a', integral indices i, j and the phase (1 or -1)
- We want to make sure computing those elements is reasonably fast.

CI String representation

- Generally, a string of K orbitals can be represented using a K-bit integer.
- This is an example of a *perfect* hash - each string is mapped to a unique number.
- However raw integer mapping is not a *minimal* hash - there are lots of gaps.
- What is desired is a *minimal perfect* hash s.t. all strings are mapped to a continuous integer range we can use for addressing.

CI String hashing

- For now we shall assume that we are dealing with at most 32 orbital strings.
- Generate all valid strings in lexicographical order.
- Observe that there are fast-changing (upper) and slow-changing (lower) halves.
- Furthermore observe that upper half are the same for any given lower half.
- Then we can construct a hash that uses upper half to lookup displacement of its lower half and the lower half to lookup the final index.
- For a 32-bit string that requires two lookup tables , 2^{16} elements each (512kb).
- `hash(s) = U[16 >> s] + L[0xFFFF & s]`

CI String example

```
std::vector<int> O, E(1); // reserve k->k excitation
for (int l = 0; l < I.size(); ++l) {
    if (I[l]) O.push_back(l); // occ. orbs
    if (!I[l]) E.push_back(l); // exc. orbs
}
for (auto o = O.begin(); o < O.end(); ++o) {
    int k = *o;
    E[0] = k; // k->k
    for (auto e = E.begin(); e < E.end(); ++e) {
        int l = *e;
        String J = I.swap(k, l);
        if (!ci.test(J,S)) continue;
        double sgn_kl = sgn(I, k, l);
        int kl = index(k, l);
        ...
```

CI String manipulations

- Use `std::bitset` and all its goodness
- Swap two indices i, j:
  `bitset.flip(i); bitset.flip(j);`
- Compute phase of $i, j, i < j$:
  clear hi bits: `s = s & ((1<<j)-1)`
  clear lo bits: `s = s >> i; s = s >> 1;`
  compute set bits: `p = biset(s).count() OR _mm_popcnt(s)`
  even phase is 1, odd is -1:
  `return (ij % 2) ? -1 : 1;`
- Test if the CI determinant is allowed. Whaaaa .. ?

Restricted CI

- Number of the determinants can be reduced by dropping some.
- In restricted CI, the determinants differing from reference by some factor are dropped.
- For example in (4 4) CISD, determinant pair (1100,1010) is allowed but (0110,0011) is not.
- To get the excitation of a string from reference, we need string distance.
- With bit strings, it is easy by XOR string with reference: `bitset(r ^ s).count()/2;`
- And total rank/excitation is the sum of alpha and beta rank/excitations

Overall CI algorithm

- CI algorithm can be broken down into two parts: the calculation of σ vector, $\sigma = \mathbf{H}c$ and Davidson iteration
- σ computation is computationally and I/O heavy, especially σ_3 part
- The Davidson step is a straightforward linear algebra, primarily dot products and scaling.

Sigma 1/2

- Sigma 2 computations can be formulated as $\sigma(a,b) = \sum_{b'} F(b \to b')C(a,b')$, where $F(b \to b')$ depends only on beta string excitation and corresponding one- and two- electron integrals. Sigma 1 for a' is defined similarly
- In the closed-shell case, $C(a,b)$ is symmetric and $\sigma_1(a,b) = -1^S\sigma_2(b,a)$
- The expression can be rewritten as $\sigma(b,a) = \sum_{b'} F(b \to b')C(b',a)$
- In other words, to compute column a of σ, we only need the corresponding C column
- Very easy to parallelize and to distribute data, distribute data column-wise and operate on local columns of C and σ
- In case of open-shell, transpose either matrix/vector between the two steps

Code-level optimization

- Each processor has theoretical peak of performing operations, in our case floating-point operations.
- For the purposes of chemistry we use double precision (8-byte/64-bit) floating point numbers (versus 4-byte single precision numbers.
- We can know the peak performance of double precision operations if we know Instructions Per Cycle (IPC) rate. With modern x86-64 processors, double precision IPC is 4.
- We then multiply IPC by processor frequency to get the Floating Operations Per Second (FLOPs).

Code-level optimization

- One of the biggest direct factors affecting IPC are Single Instruction Multiple Data (SIMD) instructions, also called vector instructions.
- Vector instructions apply the same operator (eg multiply, add) to a `vector` of operands
- SSE vector length is 128 bits, AVX is 256 bits.
  128 bit SIMD can evaluate two sets of 64-bit operands for the price of one.
- To make use of SIMD instructions, code must operate on contiguous chunks of data, eg:

```
for (int i = 0; i < N; ++i) {
    sigma[i] = f*C[i];
}
```

Indirect addressing

- Any sort of indirect addressing will kill code vectorization, eg:

```
for (int i = 0; i < N; ++i) {
    sigma[i] = f*C[index[i]];
}
```

- Guess what a naive CI kernel looks like?
- One solution is to transpose data
- Another solution is to factor out loops with indirect addressing and apply computed results in a loop with direct addressing.
- Yet another solution is to reorder data to avoid indirect addressing
- Or combination thereof!

Aliasing

- Other factor that kills vectorization is pointer aliasing, eg:

```
double *sigma = ...;
const double *C = ...;
for (int i = 0; i < N; ++i) {
    // compiler must assume sigma might point to C!!!
    sigma[i] = f*C[i];
}
```

- Solution is to tell compiler the pointer is not "connected" to other pointers

```
double *__restrict__ sigma = ...;
const double *__restrict__ C = ...;
for (int i = 0; i < N; ++i) {
    sigma[i] = f*C[i];
}
```

Sigma revisited

Recall in $\sigma(a,b) = \sum_{a'} F(a \to a')C(a',b)$, $F(b \to a')$ depends only on alpha string and corresponding one- and two- electron integrals.

```
range rb = ...; // some beta subrange
Matrix c = C(alpha, rb).transpose();
Matrix s(rb.size, alpha.size());
for (int i = 0; i < alpha.size(); ++i) {
    Vector F = Vector::Zero(alpha.size());
    // all excitation from alpha[i]
    F = sigma2(alpha[i], alpha, H, V);
    for (int k = 0; k < alpha.size(); ++k) {
        double f = F(k);
        if (fabs(f) < 1e-14) continue;
        s.col(i) += f*c.col(k);
    }
}
```

Sigma 3

- $\sigma_3(a,b) = \sum_{a'b'} F(a \to a', b \to b') C(a', b')$,
- Notice the big difference from earlier σ, neither rows nor columns "match" between σ and C. That means a lot of I/O.
- Lets say we need to compute a beta column 111000. Allowed excitations are
  $111000, 110100, 110010, 110001, 101100, ...$
  All those columns will need to be loaded.
- What about a beta column 110100? Allowed excitations are
  $111000, 110100, 110010, 110001, 101100, ...$
  That's a lot of overlap with a beta column 111000.
- I/O overhead can be hidden by computing a block of beta columns at a time.

Sigma 3

The general approach I took is as follows

- Pick a block of beta columns to compute.
- For each of those beta strings, generate all valid excitations.
- Sort excitation list
- Loop over excitation list in blocks (list may have gaps)
- Load $C(:, b')$ corresponding to that block
- Evaluate $\sigma(:, b)$ columns corresponding to $C(:, b')$ block. Those $\sigma(:, b)$ columns aren't necessarily contiguous!
- Now that I/O part is (kinda) taken care of, what about computational efficiency?

Sigma 3

There is a lot of indexing involved in
$\sigma_3(a,b) = \sum_{a',b'} E_{ij}^{aa'} E_{kl}^{bb'} (ij|kl) C(a',b')$
We need to get valid excitations, phase factors, orbitals that are swapped, etc. Lets try a few tricks.

- Make a subcopy of $(ij|kl)$, lets call it $(ij|b')$, corresponding to $b \to b'$ excitations in the block.
- Apply $b \to b'$ phase factors to $(ij|b')$.
- Now the b' dimension no longer requires indirect indexing: $\sigma_3(a,b) = \sum_{a',b'} E_{ij}^{aa'} (ij|b') C(a',b')$.
- Assuming column-major storage, need to transpose matrices.
- There is no way to get rid of alpha indexing but for each alpha phase/index lookup there will be a block of vectorizable operations.

Performance

Sample computation, 124,408,576,656 determinants, 64 nodes x 16 cores, 1024 total cores.

- Overall σ time about 4300s, a bit over an hour.
- σ_{1+2} takes about 1300s
- Out of that, σ_{1+2} *kernel* accounts for about 50%. Runs at about 20% of peak.
- σ_{1+2} I/O (read and write) is about 10%.
- σ_3 takes about 3000s
- Out of that, the *kernel* accounts for about two thirds of time.
- I/O overhead isn't bad, on the order of 10% but *data reordering* is rather expensive
- With one-two expansion vectors, Davidson overhead is low, around 10% of total wall time. Davidson wall time increases with the number of expansion vectors.

Current improvements

Lexicographical ordering isn't the best.

- Consider the case of σ_3, for a rank R string, only $R-1, R, R+1$ subset is needed.
- With lexicographical ordering, the ranks are all over the place.
- I/O is less localized *and* restricted CI is difficult to integrate.
- Instead, (stable) sort the lexicographically ordered strings by their ranks.
- Modify the loops to iterate over the blocks of excitation subspaces.
- Improved data locality, integrating restricted CI is straightforward.
- Slight overhead to string indexing: the hash must assume lexicographical order.

Questions?