I/O and Memory.
Experiences from Computation Chemistry

Andrey Asadchev

VT.edu

September 30, 2013

Mountain Creek in Western Virginia

Forewarning

To keep the presentation short and accessible to a wide audience, many theoretical details are glossed over and/or represented in simplified manner.

A Very Brief Overview of Computation Chemistry

- The goal is to obtain energies and properties of a molecule from *first principles* ie via the Schrodinger equation $H\Psi = E\Psi$
- Most systems have no closed-form solution and must rely on approximations.
- The wavefunction Ψ in typically represented in terms of N atomic basis functions χ.
- The wavefunction is then a linear combination, $\psi_i = \sum_b^N c_{ib}\chi_b$
- Coefficients are optimized so as to minimize the energy.
- The most common *first-order* method is Hartree-Fock (HF), which is an iterative method.
- In its common formulation, $FC = EC$ HF requires computing so called two-electron integrals and diagonalizing the operator (eigenvalue problem)

First I/O Bottleneck

- The two-electron integrals do not change during the iterative HF steps.
- They maybe precomputed and reused, as was done in the early days.
- However, there are N^4 integrals, as opposed to F operator which is N^2, and even with the modest basis of say 750 basis functions, the storage is 2.5TB
- It was quickly realized that re-computing integrals on the fly was way more efficient.
- This approach to HF is called direct HF.
- In chemistry parlance, *direct* oftentimes refers to methods which recompute the intermediates.

Past Hartree-Fock

- The common wisdom is that HF about recovers 99% of *total* energy and produces fairly accurate geometries.
- The 1% that Hartree-Fock leaves out is what is called the correlation energy.
- The properties associated with that 1% is what theorists usually seek.
- The electron correlation methods try to recover that 1% from the HF wavefunction.

Electron correlation methods

- Moller-Plesset Second Order (MP2) is a fairly cheap method, scales as N^5.
- Coupled-Cluster (CC) methods are consider the best, but scale as N^6 and beyond.
- Configuration Interaction (CI) method gives the exact result (within the limits of the basis set) but scales as $N!$
- All electron correlation methods involve manipulations of large datasets, on the order of gigabytes and terabytes.
- Over the years researcher developed methods to decrease storage and computation, for example resolution-of-identity (RI), density-fitting (DF), and other methods. The cutting edge are the PNO methods (in my opinion).

The Problem with Memory and I/O

- The implementations typically try to store work data in memory, either on node or distributed across the cluster.
- That limits the range of machines a user can utilize and pushes a finite resource (as opposed to infinite time).
- Some data access must be fast, eg if the data is accessed in innermost loops or must be accessed in non-contiguous blocks.
- Typically one would use various tricks such as pre-fetching, loop blocking, and computation/communication overlap to hide/decrease access latency.
- But algorithms can be re-arranged and modified so that most of data is accessed only once per outer loop, in large blocks.
- In that case, the data can be stored on the filesystem with minimal overhead.

CI

- CI is given as $|\Psi\rangle = c_0\langle\Phi_0| + \sum c_i^a\langle\Phi_i^a| + ..$
- In essence, we want to consider every possible replacement from the reference.
- In other words, we want to "distribute" N electrons among K orbitals: there are $\binom{K}{N_\alpha}$ alpha and $\binom{K}{N_\beta}$ beta arrangements (strings).
- A particular configuration will be an alpha-beta string pair (determinant) and we want to consider ALL pairs.
- The entire Hamiltonian is square the number of ALL pairs!!!
- Only for smallest of values 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 (Davidson method)

- The strings can be represented with 0s (empty orbitals) and 1s (electrons).
- This leads to a bit strings representation, eg. 101..100.
- For example $K = 4, N = 2$ configuration will generate $1100, 1010, 1001, 0110, 0101, 0011$
- Bit strings can be indexed and manipulated efficiently using bit operators and hardware instruction: `&`, `>>`, `POPCNT`, ...
- The entire string list can sorted lexicographically and indexed using minimal perfect hash:
  `hash(s) = Upper[16 >> s] + Lower[0xFFFF & s]`

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 $\sigma_1, \sigma_2, \sigma_3$.

$$\sigma_1(a,b) = \sum_{b'}\sum_{ij} \langle b|E^{\beta}_{ij}|b'\rangle h'_{ij} C(a,b') + \frac{1}{2}\sum_{b'}\sum_{ijkl} \langle b|E^{\beta}_{ij}|b'\rangle\langle b'|E^{\beta}_{kl}|b''\rangle (ij|kl) C(a,b')$$

$$\sigma_2(a,b) = \sum_{a'}\sum_{ij} \langle a|E^{\alpha}_{ij}|a'\rangle h'_{ij} C(a',b) + \frac{1}{2}\sum_{a'}\sum_{ijkl} \langle a|E^{\alpha}_{ij}|a'\rangle\langle a'|E^{\alpha}_{kl}|a''\rangle (ij|kl) C(a',b)$$

$$\sigma_3(a,b) = \frac{1}{2}\sum_{a'b'}\sum_{ijkl} \langle a|E^{\alpha}_{ij}|a'\rangle\langle b|E^{\beta}_{kl}|b'\rangle (ij|kl) C(a',b')$$

The CI Dimensions

To give an idea of CI datasets

- (16 16) CI (16 orbitals, 8 alpha, 8 beta): 12,870 strings, 165 million determinants. C, σ each require 1.3GB
- (20 20) CI (20 orbitals, 10 alpha, 10 beta): 184,756 strings, 34 billion determinants. C, σ each require 273GB
- (21 20) CI (21 orbitals, 10 alpha, 10 beta): 352,716 strings, 124 billion determinants. C, σ each require 995GB.
- Moreover, previous C vectors must be retained in the Davidson expansion.

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.
- The data for the Davidson step can be made local to each node. There is no data transfer required other than the reduction of partial dot products. Very straightforward to implement to utilize filesystem to the max.
- Since the data will be local to a node, the I/O will scale with the number of nodes.

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

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.

The general approach

- 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!
- The I/O part is (kinda) taken care of.

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. One could 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%.
- σ_{1+2} I/O (read and write) is about 10%.
- σ_3 takes about 3000s, runs at about 20% of max CPU rate.
- Out of that, the *kernel* accounts for about two thirds of time.
- I/O overhead isn't bad, on the order of 10% but in-memory 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.

Further 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 and I/O is not localized
- Instead, (stable) sort the lexicographically ordered strings by their ranks.
- Now the I/O will be limited only to neighbouring blocks $R-1, R, R+1$

Programming Aspects

The object hierarchy is three-tiered

- In-core `Tensor, Matrix, Vector`, derived from `Eigen` library.
- Distributed `Array`, implemented in terms of various backends, eg ARMCI, HDF5, MPI, etc.
- `File` and file objects `Dataset, Group, ...` implemented on top of HDF5.
- Each object has `range` access operators to give sub-block access.
- In-core objects can be read/written to/from `Array` and file objects.
- `Array` object can be read/written to/from file objects.

Example

```
// create/open file
File file = File("file.h5");
// create dataset
File::Dataset<double> ds(file, "my dataset", dims);
Matrix a = Matrix::Random(m,n); // create some matrices
Matrix b = Matrix::Random(m-1,n-1);
ds << a; // write matrix to dataset
ds(range(0,m-1), range(0,n-1) >> b; // read subset
...
// create array, comm is MPI communicator
Array<double> array("my array", dims, comm);
a(0,2) = 3.14;
// write to distributed array
array(range(1,m), range(1,n)) << a;
```

Questions.

Questions.