TCE

Presented by Mark Glines
(ETI)

Outline

- What is TCE?
- What have I done?
- Some crazy math!
- Some crazy code!
- Some brainstorming...
- Related research

What is it?

- Tensor Contraction Engine
- A quantum chemistry thing
- A big feature of NWChem
- (NWChem is a quantum chemistry tool, maintained by PNNL)
- 3.1 million lines of code! it is the biggest subdirectory of nwchem/src/
- A framework for solving electrical Schrödinger equations
- A python library which turns math expressions into Fortran code
- ...and it looks like about 2.99 million of those lines of code, in 11187 files, were generated by this library

Okay. But what is it?

The name "TCE" is used variously to refer to:

- The "tce" keyword in NWChem job (input) files
- The feature enabled by that keyword
- the src/tce/ subdirectory in the NWChem sources
- The python library, tce.py, which turns math expressions into Fortran code
- A GUI which drives the whole process of generating Fortran code (consisting of several scripts and libraries)
- The generated fortran code
  - CCSD: coupled cluster singles & doubles
  - CCSDT: coupled cluster singles, doubles & triples
  - CC2: second-order approximate coupled cluster
  - MBPT2: second-order many-body perturbation theory
  - Many, many others

What have I done?

- I've added code to tce.py to generate simple, serial C code
- I've produced a simple software project which runs it the same way NWChem does
- ...and verifies the output against NWChem's outputs
- Er, I mean "simple" from a runtime / language perspective. It's still doing the same crazy math with 2-d and 4-d tensors
- And those tensors are block-sparse and symmetry-zoned, and all of that is defined by the weird data structures NWChem produces
- But the tensors live in memory, there's no networking, no threading, no filesystem tricks
- ... yet. NWChem sometimes creates the input data lazily, to make things fit, at some point we will have to do the same

So what's the result?

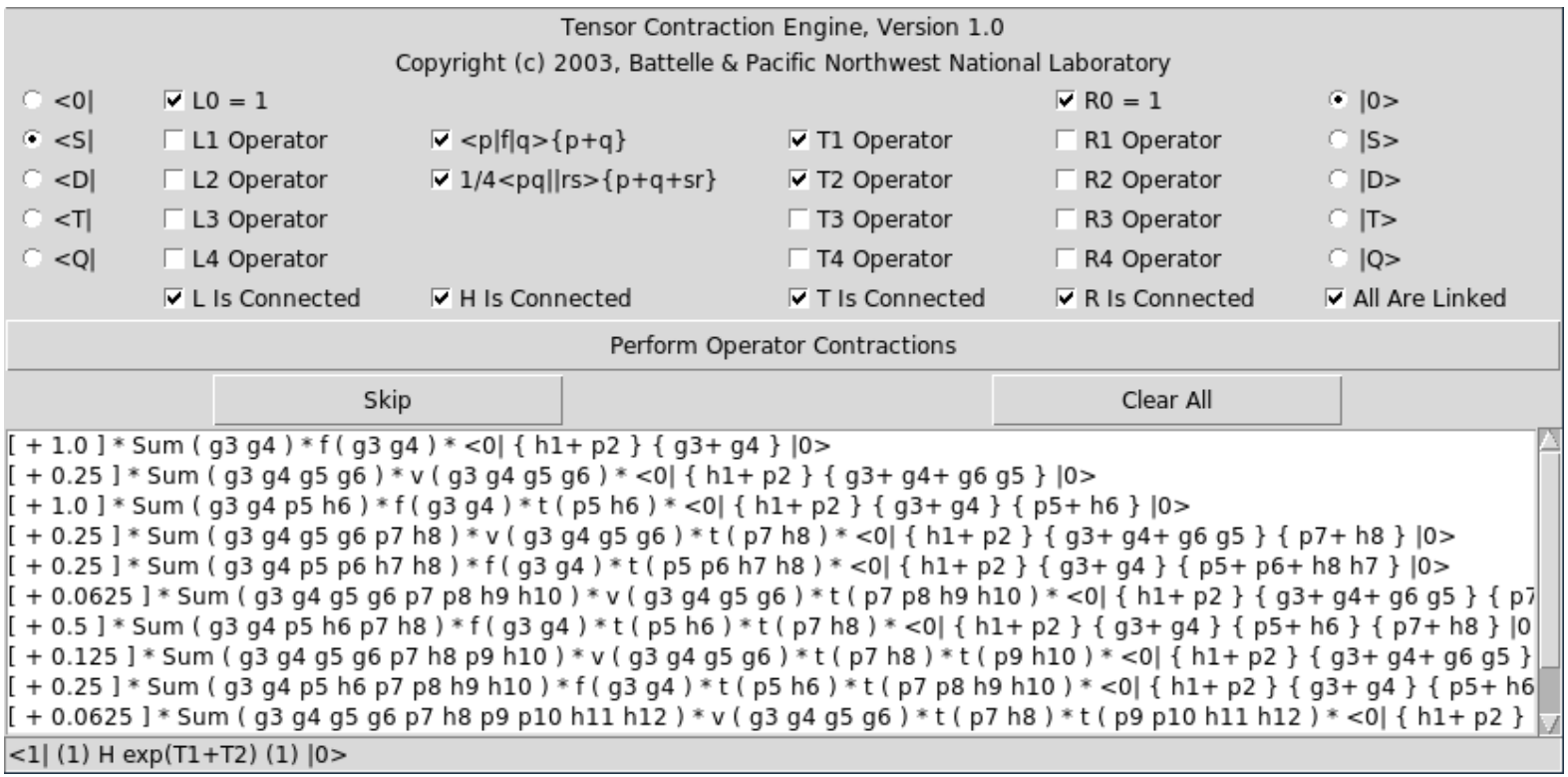
- A git repository with a Makefile, a couple of Python files, a small C file, and some compressed data files
- The C file has a main() which drives the process
- It also sets up the data structures, and implements simple versions of common NWChem functions
- The Makefile runs the python to generate C implementations of the CC2 method, decompresses the data files, builds the test program and runs it
- Hopefully, this will provide a simple, accessible way to look at TCE's performance

What does it look like to run it?

```
infinoid@forge:~/workspace/swarm/tce$ make
python driver.py cc2_t1 cc2_t1.tt >/dev/null
gcc -g -Wall -DUSE_ATLAS_BLAS    -c -o cc2_t1.o cc2_t1.c
python driver.py cc2_t2 cc2_t2.tt >/dev/null
gcc -g -Wall -DUSE_ATLAS_BLAS    -c -o cc2_t2.o cc2_t2.c
gcc -g -Wall -DUSE_ATLAS_BLAS    -c -o cc2_t1_t2_standalone.o cc2_t1_t2_standalo
ne.c
gcc -L/usr/lib/atlas-base -Wl,-rpath,/usr/lib/atlas-base  -o cc2_t1_t2_standalon
e cc2_t1.o cc2_t2.o cc2_t1_t2_standalone.o -lcblas
lzma -d -k data/test_cc2/params.i1.dump.lzma
lzma -d -k data/test_cc2/f1.i1.before.dump.lzma
lzma -d -k data/test_cc2/v2.i1.before.dump.lzma
lzma -d -k data/test_cc2/t1.i1.before.dump.lzma
lzma -d -k data/test_cc2/t2.i1.before.dump.lzma
lzma -d -k data/test_cc2/r1.i1.after.dump.lzma
lzma -d -k data/test_cc2/r2.i1.after.dump.lzma
./cc2_t1_t2_standalone data/test_cc2/params.i1.dump data/test_cc2/f1.i1.before.d
ump data/test_cc2/v2.i1.before.dump data/test_cc2/t1.i1.before.dump data/test_cc
2/t2.i1.before.dump data/test_cc2/r1.i1.after.dump data/test_cc2/r2.i1.after.dum
p
0 errors detected in total.
infinoid@forge:~/workspace/swarm/tce$
```

TCE math, badly mangled by a lowly software engineer

- TCE produces code that iteratively solves expressions.
- TCE expressions have the general form:
  $\langle bra| \hat{L}\hat{H}e^{\hat{T}}\hat{R} |ket\rangle$
- They have a GUI which allows you to select which tensors are actually present, and what form they take
- For example, the cc2_t1 problem has these boxes checked:



Where it sits in the overall process

- TCE takes that set of checkboxes, and generates crazy math
- It then generates code, which you call iteratively
- The goal (for cc2_t1 and cc2_t2 at least) is to find the right values of T
- The code generates a tensor full of residuals
- The outer loop calls it and adds the residuals back into T
- In the previous slide, the T1 and T2 boxes were checked
- That means we have T1 (a 2D tensor) and T2 (a 4D tensor)
- ...and it means e is raised to the power of $(T1 + T2)$
- cc2_t1 generates 2D tensor R1, which gets added back into T1
- cc2_t2 generates 4D tensor R2, which gets added back into T2
- Both functions take T1 and T2 as inputs, as well as the Fock matrix F1 (a 2D tensor generated previously by SCF) and a tensor V2 (a 4D tensor of second-order integrals)

Where it sits (continued)

- The C function has the following prototype:
- void cc2_t1(double* d_f1,double* d_i0,double* d_t1,double* d_t2,double* d_v2, int* k_f1_offset,int* k_i0_offset,int* k_t1_offset,int* k_t2_offset,int* k_v2_offset);
- d_* is a raw pointer to the data (in the C version)
- k_*_offset is a lookup table, which maps block IDs to offsets (these things are sparse)
- "i0" is the 2D residuals tensor that this function outputs; the output gets added to T1
- cc2_t2 is similar, except that "i0" there is a 4D residuals tensor which gets added to T2
- The implementation is broken out into subroutines, one subroutine per line of math

$$z^{h_7}_{p_8} = f^{h_7}_{p_8} \qquad \text{(cc2\_t1\_2\_2\_1)}$$
$$+ t^{p_5}_{h_6} v^{h_6 h_7}_{p_5 p_8} \qquad \text{(cc2\_t1\_2\_2\_2)}$$

$$x^{h_7}_{h_1} = f^{h_7}_{h_1} \qquad \text{(cc2\_t1\_2\_1)}$$
$$+ t^{p_8}_{h_1} z^{h_7}_{p_8} \qquad \text{(cc2\_t1\_2\_2)}$$
$$- t^{p_4}_{h_5} v^{h_5 h_7}_{h_1 p_4} \qquad \text{(cc2\_t1\_2\_3)}$$
$$- \frac{1}{2} t^{p_3 p_4}_{h_1 h_5} v^{h_5 h_7}_{p_3 p_4} \qquad \text{(cc2\_t1\_2\_4)}$$

$$y^{p_2}_{p_3} = f^{p_2}_{p_3} \qquad \text{(cc2\_t1\_3\_1)}$$
$$- t^{p_4}_{h_5} v^{h_5 p_2}_{p_3 p_4} \qquad \text{(cc2\_t1\_3\_2)}$$

$$r^{p_2}_{h_1} = f^{p_2}_{h_1} \qquad \text{(cc2\_t1\_1)}$$
$$- t^{p_2}_{h_7} x^{h_7}_{h_1} \qquad \text{(cc2\_t1\_2)}$$
$$+ t^{p_3}_{h_1} y^{p_2}_{p_3} \qquad \text{(cc2\_t1\_3)}$$
$$- t^{p_3}_{h_4} v^{h_4 p_2}_{h_1 p_3} \qquad \text{(cc2\_t1\_4)}$$

...

What do the expression subroutines look like?

```
1479 void cc2_t1_1(double* d_a,double* d_c,int* k_a_offset,int* k_c_offset) { // tce.py:11781
1480     //$Id: tce.py,v 1.10 2002/12/01 21:37:34 sohirata Exp $
1481     //This is a ISOC99 program generated by Tensor Contraction Engine v.1.0.ETI
1482     //Copyright (c) Battelle & Pacific Northwest National Laboratory (2002)
1483     /* ElementaryTensorContraction:
1484      * i0 ( p2 h1 )_f + = 1 * f ( p2 h1 )_f
1485      */ // tce.py:6358
1486     int p2b, h1b, dimc, p2b_1, h1b_1, dim_common, dima_sort, dima; // tce.py:11803
1487     double* k_a_sort, * k_a, * k_c; // tce.py:11803
1488     for(p2b = noab;p2b < noab+nvab;p2b++) { // tce.py:12147
1489         for(h1b = 0;h1b < noab;h1b++) { // tce.py:12145
1490             if(!((!restricted) || (k_spin[p2b]+k_spin[h1b] != 4))) continue; // tce.py:12198
1491             if(!(k_spin[p2b] == k_spin[h1b])) continue; // tce.py:12250
1492             if(!((k_sym[p2b]^k_sym[h1b]) == irrep_f)) continue; // tce.py:12305
1493             dimc = k_range[p2b] * k_range[h1b]; // tce.py:6648
1494             tce_restricted_2(p2b,h1b,&p2b_1,&h1b_1); // tce.py:6687
1495             dim_common = 1; // tce.py:6722
1496             dima_sort = k_range[p2b] * k_range[h1b]; // tce.py:6735
1497             dima = dim_common * dima_sort; // tce.py:6740
1498             if(!(dima > 0)) continue; // tce.py:6768
1499             k_a_sort = tce_double_malloc(dima); // tce.py:6775
1500             k_a = tce_double_malloc(dima); // tce.py:6781
1501             tce_get_hash_block(d_a,k_a,dima,k_a_offset,(h1b_1 + (noab+nvab) * (p2b_1))); // tce.py:6916
1502             tce_sort_2(k_a,k_a_sort,k_range[p2b],k_range[h1b],1,0,1.0); // tce.py:6933
1503             tce_free(k_a); // tce.py:6946
1504             k_c = tce_double_malloc(dimc); // tce.py:7315
1505             tce_sort_2(k_a_sort,k_c,k_range[h1b],k_range[p2b],1,0,1.0); // tce.py:7481
1506             tce_add_hash_block(d_c,k_c,dimc,k_c_offset,(h1b + noab * (p2b - noab))); // tce.py:7519
1507             tce_free(k_c); // tce.py:7528
1508             tce_free(k_a_sort); // tce.py:7538
1509         } // tce.py:12150
1510     } // tce.py:12150
1511 } // tce.py:11816
```

What were all those for-loops and if-statements?

- The for-loops are over block-columns and block-rows
- *noab* and *nvab* set the number of block rows and columns
- *k_sym* and *k_spin* set the spatial and spin symmetry domains
- *k_range* defines the number of rows and columns in a block

k_spin	k_sym	k_range	Row-ID	1 1 0 11 8	1 0 12 9	1 1 8 10	1 2 11 11	1 3 18 12	2 0 11 13	2 0 12 14	2 1 8 15	2 2 11 16	2 3 18 17
1	0	4	0	44	48								
1	1	1	1			8							
1	2	1	2				11						
1	3	3	3					54					
2	0	4	4										
2	1	1	5										
2	2	1	6										
2	3	3	7										

noab = 8, nvab = 10

How can we improve the serial performance?

- tce.py has already done a lot of work to optimize the math
  - It reduces algorithmic complexity, and redundant computation, by reusing intermediate values
  - It also applies cost models to minimize computation and memory footprints
- That said, it does not always generate the smartest *code*
  - cc2_t1_1 transposes the input, just to transpose it back
  - It also mallocs, frees and copies more than it needs to
- It's also calling non-optimized library routines, which are cheap knockoffs of the Fortran/NWChem versions
- The standard software engineering tricks should apply here
  - Making the code as vectorizable as possible
  - Reducing inner loop logic
  - Reducing data movement
  - Reusing buffers

How can we improve the scalability?

- Well, I think there is a lot of parallelism here
- If you look at the math, the output is a sum of separate pieces
- Those pieces can be calculated independently, and can be summed in parallel
- Some of those pieces are, themselves, sums of other pieces
- So you can look at it as a data dependency DAG
- If you look at the implementation, the various tensors are broken into blocks too
- Separate blocks can be worked on separately, or decomposed further
- If the data grows too large to fit onto a single compute node, we can start to distribute that as well
- The "v" tensor, in particular, can be quite large
- There may be gains from splitting that tensor across compute nodes

What's already been done?

- Well, NWChem is doing parallelism its own way, of course
  - It executes the expression functions one at a time, in order
  - Points in the iterator-space are assigned to compute nodes in a round-robin fashion
  - Every compute node does the outer set of for-loops, and uses a "NXTVAL()" function to decide whether to skip the work
  - There is a reduction at the end of each expression function, where the partial sums are merged
  - (This is my interpretation of the Fortran code, any inaccuracies here are my fault)
- There was also a paper at SC13 related to TCE
  - "A framework for load balancing of tensor contraction expressions via dynamic task partitioning"
  - `http://dl.acm.org/citation.cfm?id=2503290`
  - They did some interesting things with the iteration-space
- There's probably more in the literature, I haven't done a full search yet

Take it. Use it. Make TCE fast.

- Our code: `https://xstack.etinternational.com/git/tce`
- TCE is here: `http://www.csc.lsu.edu/~gb/TCE/`
- NWChem is here: `http://www.nwchem-sw.org/`
- This is the file in NWChem which calls cc2_t1 and cc2_t2:
  `https://svn.pnl.gov/svn/nwchem/trunk/src/tce/ccsd_energy_loc.F`