

Algorithms and programming models for coupled-cluster methods

Jeff Hammond, Edgar Solomonik, Devin Matthews,
David Ozog, Eugene DePrince

Argonne (LCF), UC Berkeley (CS), UTexas (Chem),
UOregon (CS), Georgia Tech (Chem)

15 August 2012



Argonne
NATIONAL
LABORATORY

*With great (computing) power
comes great (programmer) responsibility*

- Stan Lee (Spiderman's Uncle Ben)
or Voltaire or Luke 12:48 (sort of)

A dream doesn't become reality through magic; it takes sweat, determination and hard work.

- Colin Powell

Since when is parallel programming become exempt from common sense rules about how life works?

Rewarding programming models

PM	Hello	NAS	Application or Library				Example
			100KLOC	>5Y	Peta.		
MPI	Y	Y	Y	Y	Y		QBOX
MPI/OpenMP	Y	Y	Y	Y	Y		CP2K
MPI/CUDA	Y	Y	Y	Y	Y		LAMMPS
MPI/Pthread	Y	?	Y	Y	Y		MPQC
MPI/CAF/OMP	Y	?	Y	N	Y		GTS
Global Arrays	Y	Y	Y	Y	Y		NWChem
ARMCI	Y	Y	N	N	Y		Scafacos
Charm++	Y	Y	Y	Y	Y		NAMD
OpenMP	Y	Y	Y	Y	-		MKL
CUDA	Y	Y	Y	Y	-		Terachem
OpenCL	N	?	?	N	-		WARPM
CAF, UPC*	Y	Y	?	?	?		?
Chapel, X10	Y	Y	?	?	?		?
ParalleX	Y	Y	?	?	?		?

* SHMEM, too.

Coupled-cluster theory

Coupled-cluster theory

The coupled-cluster (CC) wavefunction ansatz is

$$|CC\rangle = e^T |HF\rangle$$

where $T = T_1 + T_2 + \dots + T_n$.

T is an excitation operator which promotes n electrons from occupied orbitals to virtual orbitals in the Hartree-Fock Slater determinant.

Inserting $|CC\rangle$ into the Schrödinger equation:

$$\hat{H}e^T |HF\rangle = Ecc e^T |HF\rangle \quad \hat{H}|CC\rangle = Ecc |CC\rangle$$

Coupled-cluster theory

$$|CC\rangle = \exp(T)|0\rangle$$

$$T = T_1 + T_2 + \cdots + T_n \quad (n \ll N)$$

$$T_1 = \sum_{ia} t_i^a \hat{a}_a^\dagger \hat{a}_i$$

$$T_2 = \sum_{ijab} t_{ij}^{ab} \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_j \hat{a}_i$$

$$|\Psi_{CCD}\rangle = \exp(T_2)|\Psi_{HF}\rangle$$

$$= (1 + T_2 + T_2^2)|\Psi_{HF}\rangle$$

$$|\Psi_{CCSD}\rangle = \exp(T_1 + T_2)|\Psi_{HF}\rangle$$

$$= (1 + T_1 + \cdots + T_1^4 + T_2 + T_2^2 + T_1 T_2 + T_1^2 T_2)|\Psi_{HF}\rangle$$

Coupled-cluster theory

Projective solution of CC:

$$\begin{aligned} E_{CC} &= \langle HF | e^{-T} H e^T | HF \rangle \\ 0 &= \langle X | e^{-T} H e^T | HF \rangle \quad (X = S, D, \dots) \end{aligned}$$

CCD is:

$$\begin{aligned} E_{CC} &= \langle HF | e^{-T_2} H e^{T_2} | HF \rangle \\ 0 &= \langle D | e^{-T_2} H e^{T_2} | HF \rangle \end{aligned}$$

CCSD is:

$$\begin{aligned} E_{CC} &= \langle HF | e^{-T_1 - T_2} H e^{T_1 + T_2} | HF \rangle \\ 0 &= \langle S | e^{-T_1 - T_2} H e^{T_1 + T_2} | HF \rangle \\ 0 &= \langle D | e^{-T_1 - T_2} H e^{T_1 + T_2} | HF \rangle \end{aligned}$$

Notation

$$\begin{aligned} H &= H_1 + H_2 \\ &= F + V \end{aligned}$$

F is the Fock matrix. CC only uses the diagonal in the canonical formulation.

V is the fluctuation operator and is composed of two-electron integrals as a 4D array.

V has 8-fold permutation symmetry in V_{pq}^{rs} and is divided into six blocks: V_{ij}^{kl} , V_{ij}^{ka} , V_{ia}^{jb} , V_{ij}^{ab} , V_{ia}^{bc} , V_{ab}^{cd} .

Indices i, j, k, \dots (a, b, c, \dots) run over the occupied (virtual) orbitals.

CCD Equations

$$R_{ij}^{ab} = V_{ij}^{ab} + P(ia, jb) \left[T_{ij}^{ae} I_e^b - T_{im}^{ab} I_j^m + \frac{1}{2} V_{ef}^{ab} T_{ij}^{ef} + \frac{1}{2} T_{mn}^{ab} I_{ij}^{mn} - T_{mj}^{ae} I_{ie}^{mb} - I_{ie}^{ma} T_{mj}^{eb} + (2T_{mi}^{ea} - T_{im}^{ea}) I_{ej}^{mb} \right]$$

$$I_b^a = (-2V_{eb}^{mn} + V_{be}^{mn}) T_{mn}^{ea}$$

$$I_j^i = (2V_{ef}^{mi} - V_{ef}^{im}) T_{mj}^{ef}$$

$$I_{kl}^{ij} = V_{kl}^{ij} + V_{ef}^{ij} T_{kl}^{ef}$$

$$I_{jb}^{ia} = V_{jb}^{ia} - \frac{1}{2} V_{eb}^{im} T_{jm}^{ea}$$

$$I_{bj}^{ia} = V_{bj}^{ia} + V_{be}^{im} (T_{mj}^{ea} - \frac{1}{2} T_{mj}^{ae}) - \frac{1}{2} V_{be}^{mi} T_{mj}^{ae}$$

Turning CC into GEMM 1

Some tensor contractions are trivially mapped to GEMM:

$$I_{kl}^{ij} + = V_{ef}^{ij} T_{kl}^{ef}$$

$$I_{(kl)}^{(ij)} + = V_{(ef)}^{(ij)} T_{(kl)}^{(ef)}$$

$$I_a^b + = V_c^b T_a^c$$

Other contractions require reordering to use BLAS:

$$I_{bj}^{ia} + = V_{be}^{im} T_{mj}^{ea}$$

$$I_{bj,ia} + = V_{be,im} T_{mj,ea}$$

$$J_{bi,ja} + = W_{bi,me} U_{me,ja}$$

$$J_{bi}^{ja} + = W_{bi}^{me} U_{me}^{ja}$$

$$J_{(bi)}^{(ja)} + = W_{(bi)}^{(me)} U_{(me)}^{(ja)}$$

$$J_x^z + = W_x^y U_y^z$$

Turning CC into GEMM 2

Reordering can take as much time as GEMM in the node-level implementation (e.g. NWChem). Why?

Routine	flops	mops	pipelined
GEMM	$O(mnk)$	$O(mn + mk + kn)$	yes
reorder	0	$O(mn + mk + kn)$	no

Increased memory bandwidth on GPU makes reordering less expensive (compare matrix transpose).

Coupled cluster on GPUs

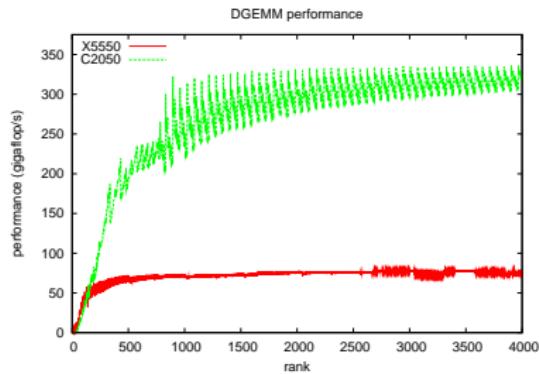
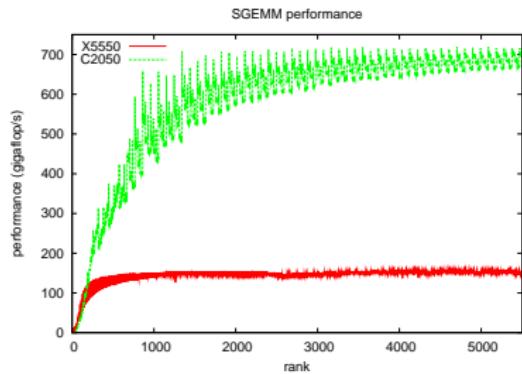
Hardware Details

	CPU		GPU	
	X5550	2 X5550	C1060	C2050
processor speed (MHz)	2660	2660	1300	1150
memory bandwidth (GB/s)	32	64	102	144
memory speed (MHz)	1066	1066	800	1500
ECC available	yes	yes	no	yes
SP peak (GF)	85.1	170.2	933	1030
DP peak (GF)	42.6	83.2	78	515
power usage (W)	95	190	188	238

Note that power consumption is apples-to-oranges since CPU does not include DRAM, whereas GPU does.

Relative Performance of GEMM

GPU versus SMP CPU (8 threads):



Maximum:
CPU = 156.2 GF
GPU = 717.6 GF

Maximum:
CPU = 79.2 GF
GPU = 335.6 GF

We expect roughly 4-5 times speedup based upon this evaluation.

CCD Algorithm

```
Copy V and F to GPU (cudaMemcpy)
WHILE ( |dT|>eps) DO
    IF  $v_{cd}^{ab}$  fits in GPU memory THEN
        Update residual (cudaDgemm)
    ELSE
        FOR all tiles DO
            copy  $v_{cd}^{ab}$  to GPU (cudaMemcpy)
            contract  $v_{ef}^{ab} t_{ij}^{ef}$  (cudaDgemm)
        END DO
        Update residual with remaining terms (cudaDgemm)
    END IF
    Update T with residual
    Compute |dT| (cudaDnrm2)
END WHILE
Copy amplitudes from GPU (cudaMemcpy)
E = t.v
```

CCD Performance Results

	Iteration time in seconds					
	our DP code			X5550		
	C2050	C1060	X5550	Molpro	TCE	GAMESS
C ₈ H ₁₀	0.3	0.8	1.3	2.3	5.1	6.2
C ₁₀ H ₈	0.5	1.5	2.5	4.8	10.6	12.7
C ₁₀ H ₁₂	0.8	2.5	3.5	7.1	16.2	19.7
C ₁₂ H ₁₄	2.0	7.1	10.0	17.6	42.0	57.7
C ₁₄ H ₁₀	2.7	10.2	13.9	29.9	59.5	78.5
C ₁₄ H ₁₆	4.5	16.7	21.6	41.5	90.2	129.3
C ₂₀	8.8	29.9	40.3	103.0	166.3	238.9
C ₁₆ H ₁₈	10.5	35.9	50.2	83.3	190.8	279.5
C ₁₈ H ₁₂	12.7	42.2	50.3	111.8	218.4	329.4
C ₁₈ H ₂₀	20.1	73.0	86.6	157.4	372.1	555.5

CCSD Equations

$$t_{ij}^{ab} d_{ij}^{ab} = v_{ij}^{ab} + P(ia, jb) R_{ij}^{ab}$$

$$R_{ij}^{ab} + = \frac{1}{2} v_{ef}^{ab} C_{ij}^{ef}$$

$$R_{ij}^{ab} + = \frac{1}{2} c_{mn}^{ab} I_{ij}^{mn}$$

$$R_{ij}^{ab} - = t_{mj}^{ae} I_{ie}^{mb} + I_{ie}^{ma} t_{mj}^{eb}$$

$$R_{ij}^{ab} + = (2t_{mi}^{ea} - t_{im}^{ea}) I_{ej}^{mb}$$

$$R_{ij}^{ab} + = t_i^e I_{ej}^{ab}$$

$$R_{ij}^{ab} - = t_m^a I_{ij}^{mb}$$

$$R_{ij}^{ab} + = t_{ij}^{ae} I_e^b$$

$$R_{ij}^{ab} - = t_{lm}^{ab} I_j^m$$

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

$$I_{jb}^{ia} = v_{jb}^{ia} - \frac{1}{2} v_{eb}^{im} (t_{jm}^{ea} + 2t_j^e t_m^a) + v_{eb}^{ia} t_j^e - v_{jb}^{im} t_m^a$$

$$I_{ci}^{ab} = v_{ci}^{ab} - v_{ci}^{am} t_m^b - v_{ci}^{mb} t_m^a$$

$$I_{jk}^{ia} = v_{jk}^{ia} + v_{ef}^{ia} C_{jk}^{ef}$$

$$I_a^i = (2v_{ae}^{im} - v_{ea}^{im}) t_m^e$$

$$I_j^i = I_j^i + I_e^i t_j^e$$

$$I_j^{ii} = (2v_{je}^{im} - v_{ej}^{im}) t_m^e + (2v_{ef}^{mi} - v_{ef}^{im}) t_{mj}^{ef}$$

$$I_b^a = (2v_{be}^{am} - v_{be}^{ma}) t_m^e - (2v_{eb}^{mn} - v_{be}^{mn}) C_{mn}^{ea}$$

$$I_{bj}^{ia} = v_{bj}^{ia} - \frac{1}{2} v_{be}^{im} (t_{mj}^{ae} + 2t_m^a t_j^e) + v_{be}^{ia} t_j^e - v_{bj}^{im} t_m^a + \frac{1}{2} (2v_{be}^{im} - v_{eb}^{im}) t_{mj}^{ea}$$

$$t_i^a d_i^a = f_i^a + R_i^a$$

$$R_i^a + = I_e^a t_i^e$$

$$R_i^a - = I_i^m t_m^a$$

$$R_i^a + = I_e^a (2t_{mi}^{ea} - t_{im}^{ea})$$

$$R_i^a + = (2v_{ei}^{ma} - v_{ei}^{am}) t_m^e$$

$$R_i^a - = v_{ei}^{mn} (2t_{mn}^{ea} - t_{mn}^{ae})$$

$$R_i^a + = v_{ef}^{ma} (2t_{mi}^{ef} - t_{im}^{ef})$$

CCSD Algorithm

Guiding principles:

- Too many arrays to fit into GPU memory.
- Copy-in every iteration in CCD was not a problem
- Want multi-GPU, mixed CPU-GPU algorithms.

Design:

- Persistent buffers but push all large arrays every iteration.
- $O(N^6)$, $O(N^5)$ on GPU.
- $O(N^5)$, $O(N^4)$ on CPU.
- Dynamically schedule some diagrams each iteration to load-balance.
- Overlap computation and communication with CUDA streams (CUBLAS compatible now).

Hybrid CCSD

	Iteration time in seconds						
	Hybrid	CPU	Molpro	NWChem	PSI3	TCE	GAMESS
C ₈ H ₁₀	0.6	1.4	2.4	3.6	7.9	8.4	7.2
C ₁₀ H ₈	0.9	2.6	5.1	8.2	17.9	16.8	15.3
C ₁₀ H ₁₂	1.4	4.1	7.2	11.3	23.6	25.2	23.6
C ₁₂ H ₁₄	3.3	11.1	19.0	29.4	54.2	64.4	65.1
C ₁₄ H ₁₀	4.4	15.5	31.0	49.1	61.4	90.7	92.9
C ₁₄ H ₁₆	6.3	24.1	43.1	65.0	103.4	129.2	163.7
C ₂₀	10.5	43.2	102.0	175.7	162.6	233.9	277.5
C ₁₆ H ₁₈	10.0	38.9	84.1	117.5	192.4	267.9	345.8
C ₁₈ H ₁₂	14.1	57.1	116.2	178.6	216.4	304.5	380.0
C ₁₈ H ₂₀	22.5	95.9	161.4	216.3	306.9	512.0	641.3

We do at least twice as many flops as Molpro due to CC formalism.

More hybrid CCSD

molecule	Basis	o	v	Hybrid	CPU	Molpro	CPU	Molpro
CH ₃ OH	aTZ	7	175	2.5	4.5	2.8	1.8	1.1
benzene	aDZ	15	171	5.1	14.7	17.4	2.9	3.4
C ₂ H ₆ SO ₄	aDZ	23	167	9.0	33.2	31.2	3.7	3.5
C ₁₀ H ₁₂	DZ	26	164	10.7	39.5	56.8	3.7	5.3
C ₁₀ H ₁₂	6-31G	26	78	1.4	4.1	7.2	2.9	5.1

Molpro optimized for $v/o \gg 1$.

Our code doesn't favor any limit except $o, v \gg 1$.

Distributed-memory coupled-cluster

TCE template

Pseudocode for $R_{i,j}^{a,b} = T_{i,j}^{c,d} * V_{a,b}^{c,d}$:

```
for i,j in occupied blocks:  
    for a,b in virtual blocks:  
        for c,d in virtual blocks:  
            if symmetry_criteria(i,j,a,b,c,d):  
                if dynamic_load_balancer(me):  
                    Get block t(i,j,c,d) from T  
                    Permute t(i,j,c,d)  
                    Get block v(a,b,c,d) from V  
                    Permute v(a,b,c,d)  
                    r(i,j,c,d) += t(i,j,c,d) * v(a,b,c,d)  
                Permute r(i,j,a,b)  
                Accumulate r(i,j,a,b) block to R
```

Observations about the TCE template

- 1 Blocking get means no overlap.
(fix with double buffering but memory usage increases)
- 2 Dynamic load balancing is **global** shared counter.
(Sriram already discussed solutions)
- 3 Get+Permute of $t(i,j,c,d)/v(a,b,c,d)$ for all $(a,b)/(i,j)$.
(data-affinity + reuse or global permute)
- 4 Permute is a nasty operation.
(need fused contraction at DGEMM speed)

There are an uncountable number of missing optimizations in any scientific code. NWChem is certainly not special in this regard.

Some of these issues hurt more on Blue Gene than COTS...

TCE Template for MMM

Pseudocode for $C_j^i = A_k^i * B_j^k$:

```
for i in I blocks:  
    for j in J blocks:  
        for k in K blocks:  
            if dynamic_load_balancer(me):  
                Get block a(i,k) from A  
                Get block b(k,j) from B  
                c(i,j) += a(i,k) * b(k,j)  
            Accumulate c(i,j) block to C
```

This is clearly not the best way to do MMM!

A better way

- Adopt the TCE node kernel approach in parallel:
tensor contraction = permute + matmul.
- Parallel permute = parallel sorting = well-understood.
- Parallel matmul = well-understood.

Therefore, parallel tensor contractions are solved, up to the implementation details and future algorithm developments in sorting and matmul.

All existing TCE technology for higher-level optimizations are still valid. Our abstraction provides a much cleaner performance model for TCE to target.

Cyclops Tensor Framework

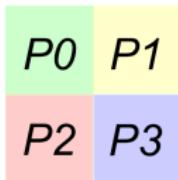
Edgar Solomonik develops CTF.

Upper-level interface and CC codes by Devin Matthews.

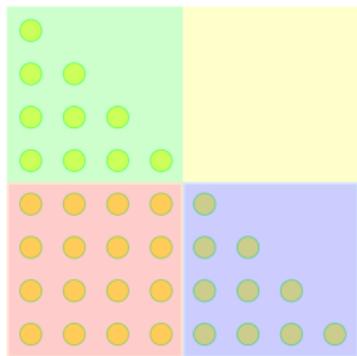
Cyclops Tensor Framework

- Can we apply the absolute state-of-the-art in dense matrix algorithms to tensors without much difficulty (and thereby capture all previous develops with respect to communication minimization, topology-awareness, numerical precision and fault-detection)?
- Can we *completely eliminate* the irregularity associated with permutation symmetry and irregular blocking that create a significant load-balancing challenge?
- Can we do it without sacrificing any of the productivity of high-level abstractions as found in TCE?

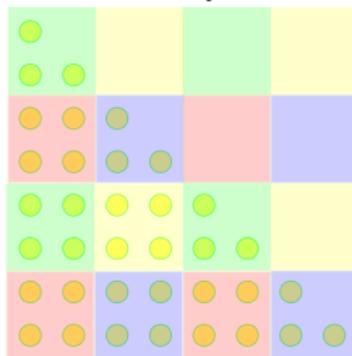
Data decompositions



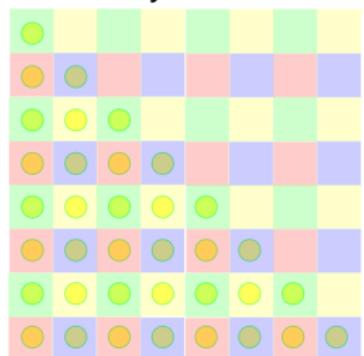
Blocked



Block-cyclic

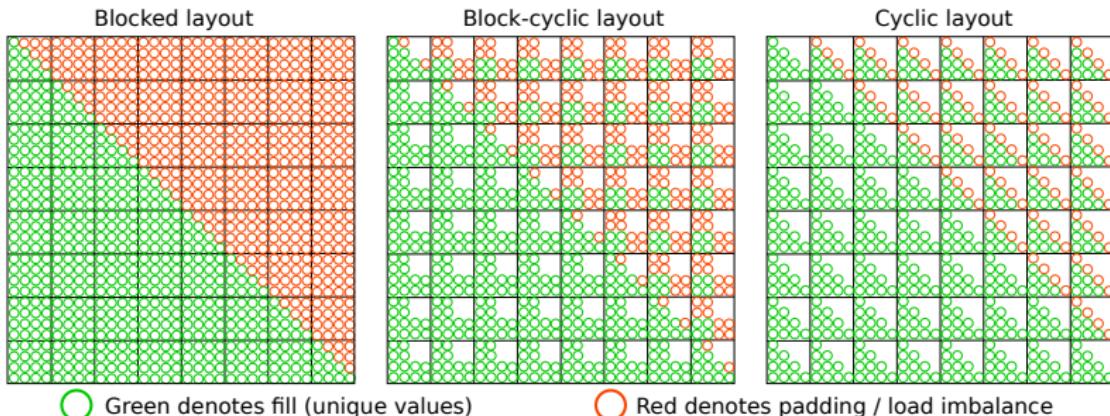


Cyclic



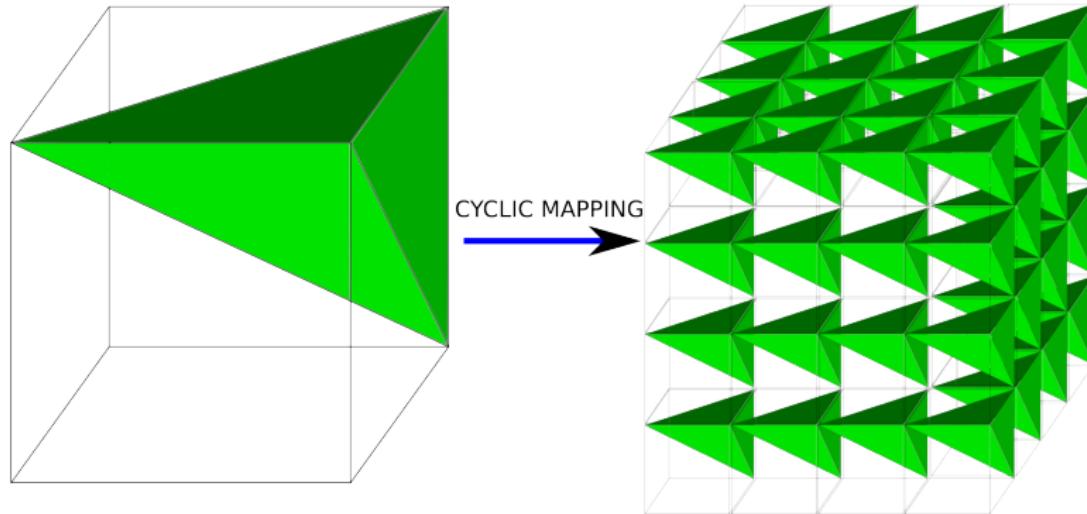
Data decompositions

Triangle of squares or square of triangles?
i.e. DGEMM vs. topo (triangle network topology anyone?)



If you fold the triangle into a rectangle, what does the *communication topology* look like?

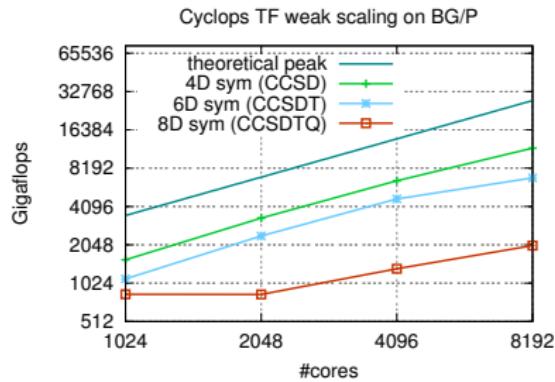
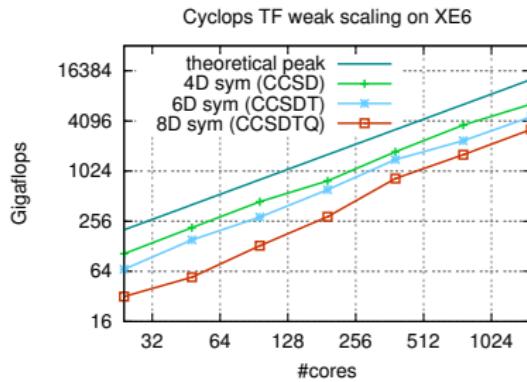
Data decompositions



Calculate memory overhead for square tiling in 4D, 6D, 8D.
Triangle-pack surface tiles and lose DGEMM...

Performance

Note: 6D and 8D are actually *more* difficult cases than found in CCSDT and CCSDTQ, but they represent that CTF can exploit all the symmetry available in both the inner and outer indices.



Have not had time to look at BGP 8D scaling issues; likely due to weird dimensions.

Science

CCD is already running. This is what the code looks like:

```
void calcE(DistTensor & T2AA,      DistTensor & T2AB,      DistTensor & T2BB,
           DistTensor & TTAA,      DistTensor & TTAB,      DistTensor & TTBB,
           DistTensor & VABIJAA,   DistTensor & VABIJAB,   DistTensor & VABIJBB,
           DistTensor & D2AA,      DistTensor & D2AB,      DistTensor & D2BB,
           DistTensor & Z2AA,      DistTensor & Z2AB,      DistTensor & Z2BB,
           DistTensor & E_CCD)
{
    TTAA["ABIJ"] = Z2AA["ABIJ"]*D2AA["ABIJ"];
    TTAB["AbIj"] = Z2AB["AbIj"]*D2AB["AbIj"];
    TTBB["abij"] = Z2BB["abij"]*D2BB["abij"];

    E_CCD = VABIJAA["ABIJ"]*TTAA["ABIJ"];
    E_CCD += VABIJAB["AbIj"]*TTAB["AbIj"];
    E_CCD += VABIJBB["abij"]*TTBB["abij"];

    TTAA["ABIJ"] -= T2AA["ABIJ"];
    TTAB["AbIj"] -= T2AB["AbIj"];
    TTBB["abij"] -= T2BB["abij"];

    T2AA["ABIJ"] += TTAA["ABIJ"];
    T2AB["AbIj"] += TTAB["AbIj"];
    T2BB["abij"] += TTBB["abij"];
}
```

Can you find the code corresponding to $E_{CCD} = V_{ij}^{ab} * T_{ij}^{ab}$?

CCD performance

$$n_o = 20, n_v = 435$$

XE6 nodes	CCSD iter. (s)
31	361.66
62	183.87
124	103.09
248	59.77

$$n_o = 20, n_v = 435$$

nodes	CCD iter. (s)
64 of XE6	74.8
512 of BGQ	37.3

This is apples-to-oranges because NWChem is doing semidirect CCSD (lower memory requirements), while CTF is doing canonical CCD. Adding singles to CTF won't change cost much.

Summary

- CTF is perfectly statically load-balanced thanks to cyclic distribution.
- CTF can immediately utilize SUMMA, Cannon, Strassen, 2.5D, etc. dense MMM.
- No automatic code generation but Devin's interface makes writing CC almost trivial. Each permutation-unique term is *one line of code*.

We're not out of the woods yet...

- Nontrivial integer computation in parallel redistribution code.
- Still exploring virtualization dimensions and thread parallelism.

Not bad given that this project began in May 2011 and is unfunded except for DOE-CSGF (Edgar and Devin).

Acknowledgments

