Scalability and programming productivity for massively-parallel wavefunction methods

Edgar Solomonik

Department of Computer Science
University of Illinois at Urbana-Champaign

American Chemical Society Annual Meeting
New Orleans, LA

March 18, 2018

Tensor notation/terminology

A *tensor* $\boldsymbol{T} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ has

- *Order* d (i.e. d *modes* / *indices*)
- *Dimensions* n_1-by-$\cdots$-by-n_d
- *Elements* $t_{i_1 \ldots i_d} = t_{\boldsymbol{i}}$ where $\boldsymbol{i} \in \bigotimes_{i=1}^{d} \{1, \ldots, n_i\}$

Tensors are multidimensional arrays with attributes

- sparsity

$$t_{ij} \neq 0 \quad \text{if} \quad (i,j) \in S$$

- symmetry or antisymmetry

$$t_{ij} = t_{ji} \quad \text{or} \quad t_{ij} = -t_{ji}$$

- algebraic structure

$$t_{ij} + t_{kl} =? \quad \text{and} \quad t_{ij} \cdot t_{kl} =?$$

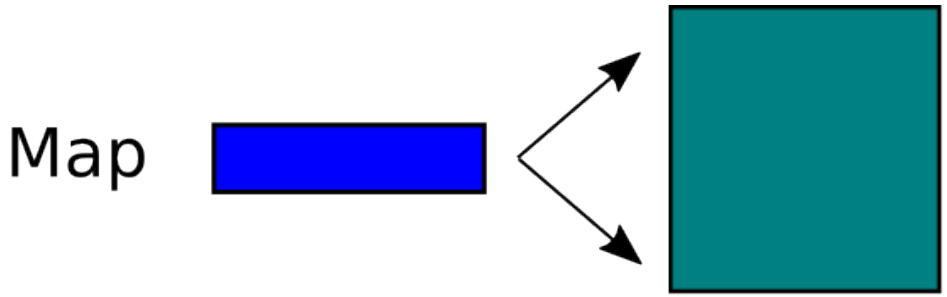
Tensors in quantum chemistry

Tensor contractions dominate cost of many wavefunction methods

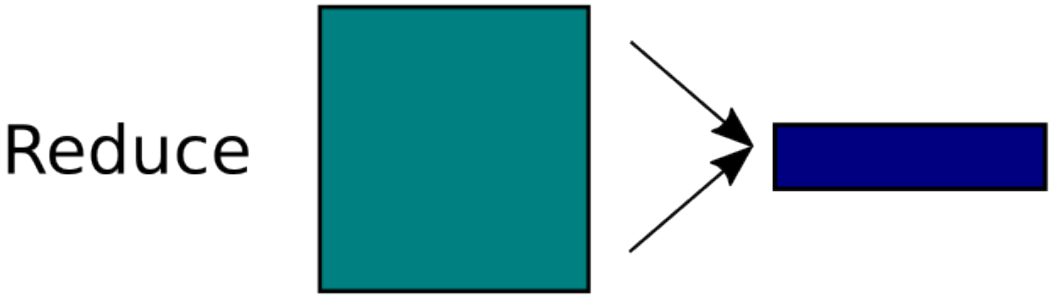
- Orbital transformations (tensor times matrix)
- *Dense tensor* contractions in Post-Hartree-Fock methods
  - Møller-Plesset perturbation
  - configuration interaction
  - coupled cluster
- *Sparse tensors*
  - localized orbitals (basis functions with compact support)
  - screening of elements
- *Tensor decompositon/factorization*
  - density fitting
  - resolution of identity

Generalized tensor summation

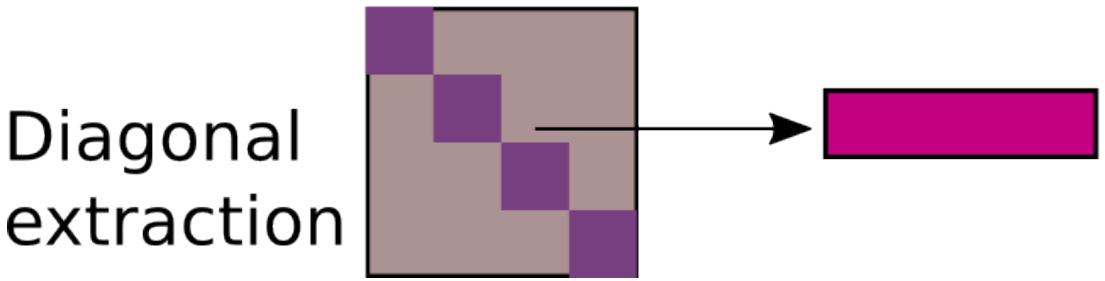
Einstein summation notation naturally expresses transformations beyond tensor contraction



$$T[i,j] = f(\mathbf{u}[i])$$



$$\mathbf{w}[i] \mathrel{+}= g(T[i,j])$$



$$\mathbf{v}[i] = h(M[i,i])$$

Generalized tensor contraction

We can identify three classes of contraction-like operations

- Tensor contraction
  - each index appears in *exactly two tensors*
  - equivalent to matrix multiplication after transposition
- Tensor contraction with *Hadamard indices*
  - an index appears in all three tensors
  - equivalent to *batched matrix multiplication* after transposition
- Tensor contractions with map/reduce
  - an index appears in only one tensor
  - amenable to pre- or post- processing via generalized summation

A stand-alone library for parallel tensor computations

Cyclops Tensor Framework (CTF)

- distributed-memory symmetric/sparse tensors as C++ objects

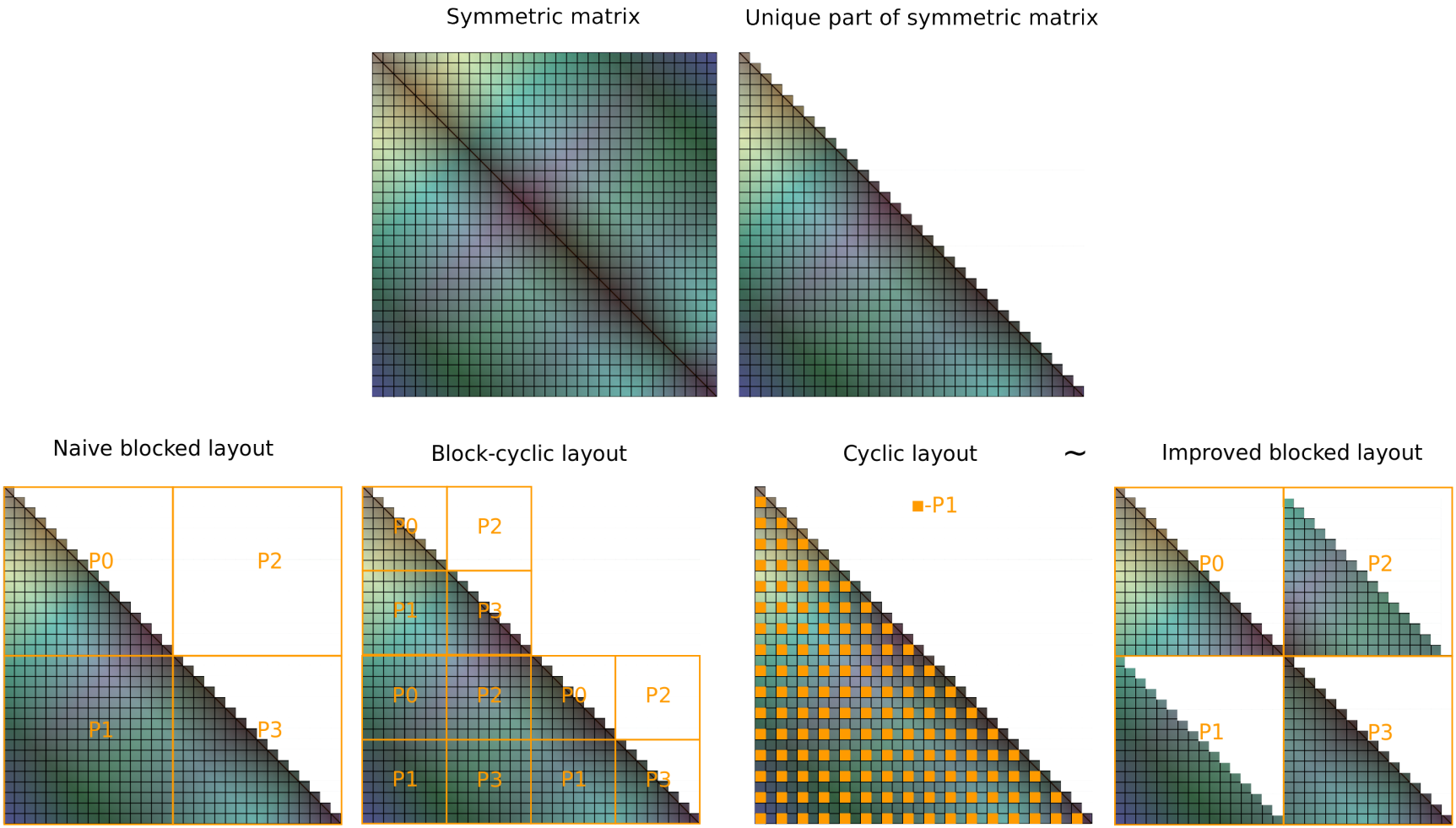
```
Matrix<int> A(n, n, AS|SP, World(MPI_COMM_WORLD));
Tensor<float> T(order, is_sparse, dims, syms, ring, world);
T.read(...); T.write(...); T.slice(...); T.permute(...);
```

- parallel generalized contraction/summation of tensors

```
Z["abij"] += V["ijab"];
B["ai"]    = A["aiai"];
T["abij"]  = T["abij"]*D["abij"];
W["mnij"] += 0.5*W["mnef"]*T["efij"];
Z["abij"] -= R["mnje"]*T3["abeimn"];
M["ij"] += Function<>([](double x){ return 1/x; })(v["j"]);
```

- NEW: Python! towards autoparallel numpy ndarray: einsum, slicing (credit to Zecheng Zhang, undergraduate, UIUC)

Symmetry and sparsity by cyclicity



for sparse tensors, a cyclic layout provides a load-balanced distribution

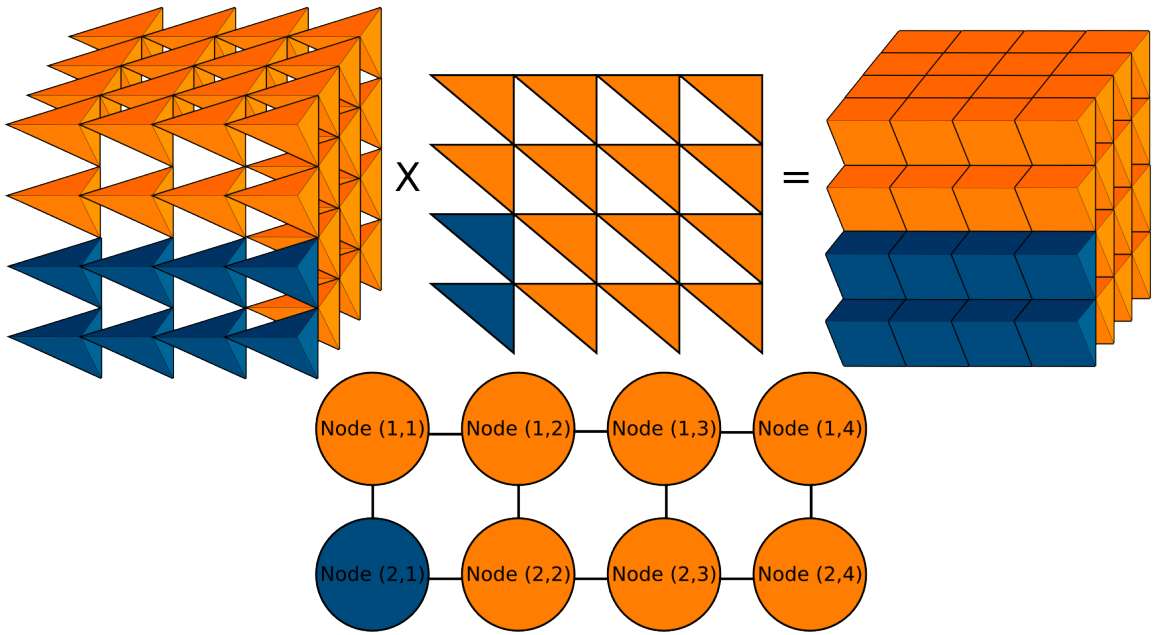
Parallel contraction in Cyclops

Cyclops uses nested parallel matrix multiplication variants

- 1D variants
  - perform a different *matrix-vector product* on each processor
  - perform a different *outer product* on each processor
- 2D variants
  - perform a different *inner product* on each processor
  - *scale a vector* on each processor then sum
- 3D variants
  - perform a different *scalar product* on each processor then sum
  - can be achieved by *nesting* 1D+1D+1D or 2D+1D or 1D+2D
- All variants are *blocked* in practice

Tensor blocking/virtualization

Preserving symmetric-packed layout using cyclic distribution constrains possible tensor blockings



Subdivision into more blocks than there are processors (virtualization)

Data mapping and redistribution

Transitions between contractions require redistribution and refolding

- 1D/2D/3D variants naturally map to 1D/2D/3D processor grids
- Initial tensor distribution is oblivious of contraction
  - by default all tensor distributed over all processors
  - user can specify any processor grid mapping
- Global redistribution done by one of three methods
  - reassign tensor blocks to processors (easy+fast)
  - reorder and reshuffle data to satisfy new blocking (fast)
  - treat tensors as sparse and sort globally by function of index
- Matricization/transposition is then done locally
  - dense tensor transpose done using HPTT (by Paul Springer)
  - sparse tensor converted to CSR sparse matrix format

Local summation and contraction

- For contractions, local summation and contraction is done via BLAS
- Threading is used via BLAS (done via OpenMP everywhere else)
- GPU offloading is available but not yet fully robust
- For sparse matrices, *MKL provides fast sparse matrix routines*
- To support general (mixed-type, user-defined) elementwise functions, manual implementations are available
- User can specify blocked implementation of their function to improve performance

Performance modeling and intelligent mapping

- Performance models used to select best contraction algorithm
- Based on *linear cost model for each kernel*

$$T \approx \underbrace{\alpha S}_{\text{latency}} + \underbrace{\beta W}_{\text{comm. bandwidth}} + \underbrace{\nu Q}_{\text{mem. bandwidth}} + \underbrace{\gamma F}_{\text{flops}}$$

- Scaling of S, W, Q, F is a function of parameters of each kernel
- Coefficients for all kernels depend on compiler/architecture
- Linear regression with Tykhonov regularization used to select parameters $\boldsymbol{x}^*$
- Model training done by benchmark suite that executes various end-functionality for growing problem sizes, collecting observations of parameters in rows of $\boldsymbol{A}$ and execution timing in $\boldsymbol{t}$

$$\boldsymbol{x}^* = \underset{\boldsymbol{x}}{\operatorname{argmin}}(||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{t}||_2 + \lambda||\boldsymbol{x}||_2)$$

Applications using Cyclops

Quantum chemistry applications

- Aquarius (lead by Devin Matthews)
- QChem via Libtensor (integration lead by Evgeny Epifanovsky)
- PySCF via new Python interface
- CC4S (lead by Andreas Grüneis and group)
- QBall (DFT code, just matrix multiplication)

Beyond quantum chemistry

- Largest-ever quantum circuit simulation (as of Oct 2017, lead by IBM and LLNL)
- Lattice QCD (lead by Bartosz Kostrzewa)
- Graph algorithms (see betweenness centrality SC 2017 paper)

Coupled cluster: an initial application driver

CCSD contractions from Aquarius (lead by Devin Matthews)

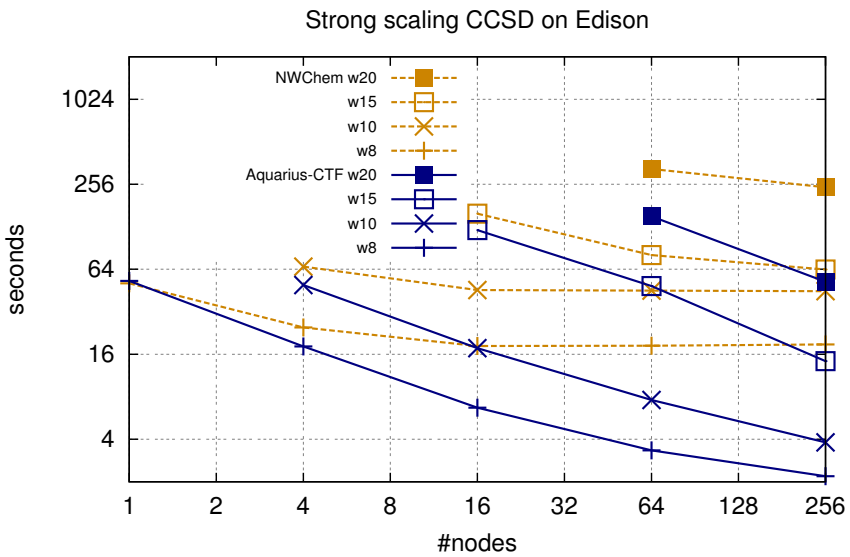
https://github.com/devinamatthews/aquarius

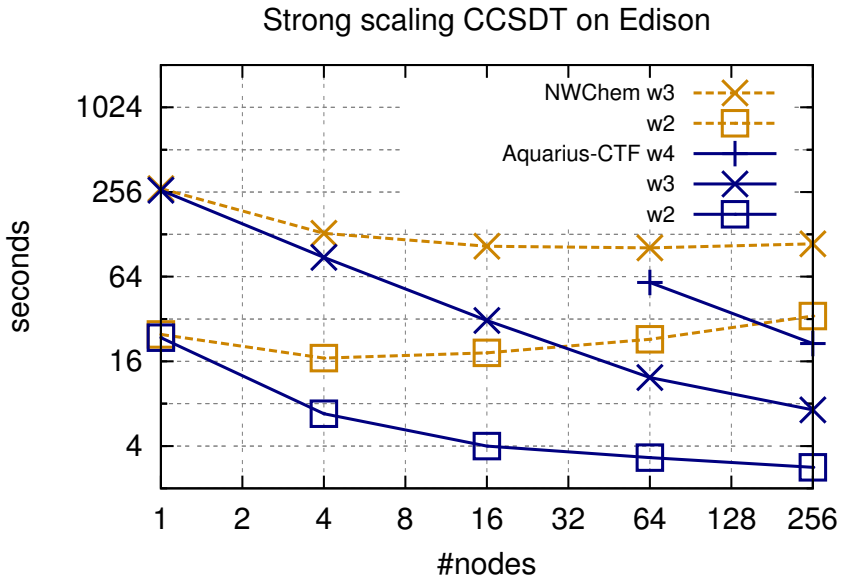
```
FMI["mi"]     += 0.5*WMNEF["mnef"]*T2["efin"];
WMNIJ["mnij"] += 0.5*WMNEF["mnef"]*T2["efij"];
FAE["ae"]     -= 0.5*WMNEF["mnef"]*T2["afmn"];
WAMEI["amei"] -= 0.5*WMNEF["mnef"]*T2["afin"];

Z2["abij"]  = WMNEF["ijab"];
Z2["abij"] += FAE["af"]*T2["fbij"];
Z2["abij"] -= FMI["ni"]*T2["abnj"];
Z2["abij"] += 0.5*WABEF["abef"]*T2["efij"];
Z2["abij"] += 0.5*WMNIJ["mnij"]*T2["abmn"];
Z2["abij"] -= WAMEI["amei"]*T2["ebmj"];
```

Comparison with NWChem

NWChem built using one-sided MPI, not necessarily best performance

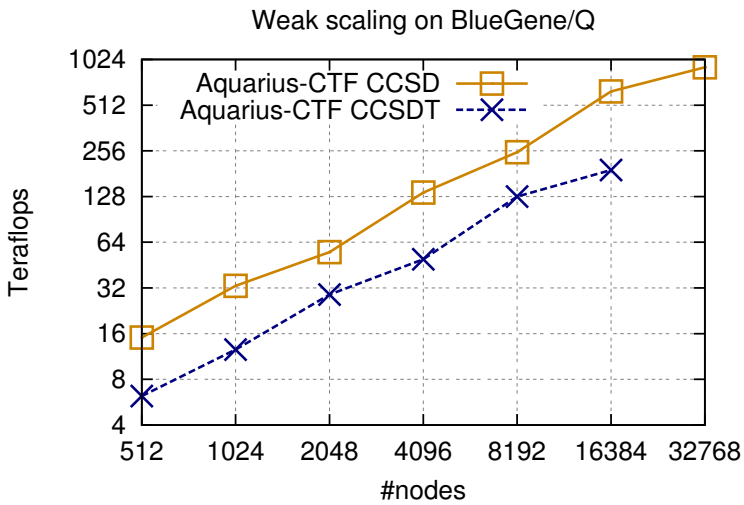
- derives equations via Tensor Contraction Engine (TCE)
- generates contractions as blocked loops leveraging (Global Arrays)

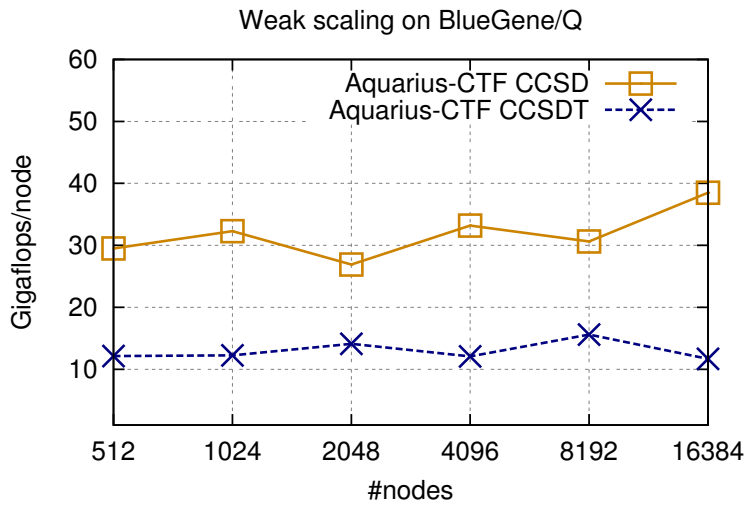


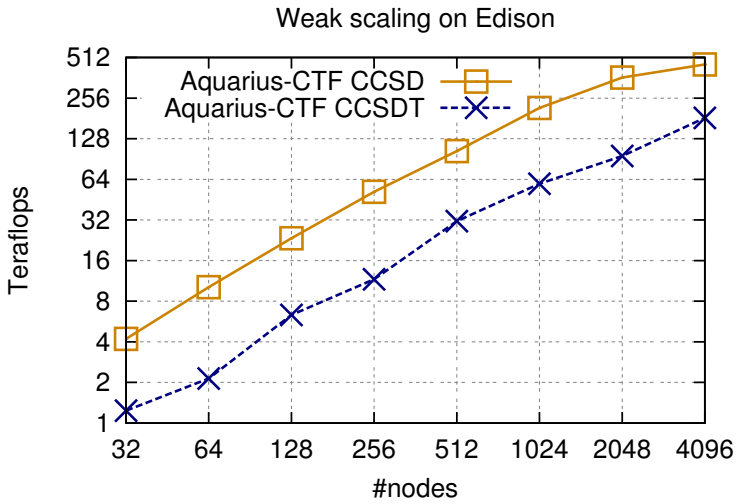


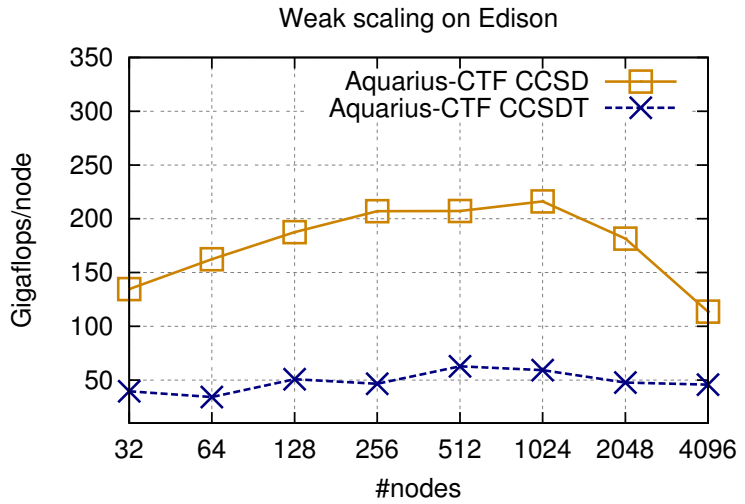
Performance of Cyclops for coupled cluster

CCSD up to 55 (50) water molecules with cc-pVDZ
CCSDT up to 10 water molecules with cc-pVDZ









MP3 method

```
Tensor<> Ea, Ei, Fab, Fij, Vabij, Vijab, Vabcd, Vijkl, Vaibj;
... // compute above 1-e an 2-e integrals

Tensor<> T(4, Vabij.lens, *Vabij.wrld);
T["abij"] = Vabij["abij"];

divide_EaEi(Ea, Ei, T);

Tensor<> Z(4, Vabij.lens, *Vabij.wrld);
Z["abij"]  = Vijab["ijab"];
Z["abij"] += Fab["af"]*T["fbij"];
Z["abij"] -= Fij["ni"]*T["abnj"];
Z["abij"] += 0.5*Vabcd["abef"]*T["efij"];
Z["abij"] += 0.5*Vijkl["mnij"]*T["abmn"];
Z["abij"] += Vaibj["amei"]*T["ebmj"];

divide_EaEi(Ea, Ei, Z);

double MP3_energy = Z["abij"]*Vabij["abij"];
```

MP3 dense division

A naive dense version of division in MP2/MP3

```
void divide_EaEi(Tensor<> & Ea,
                 Tensor<> & Ei,
                 Tensor<> & T){
  Tensor<> D(4,T.lens,*T.wrld);
  D["abij"] += Ei["i"];
  D["abij"] += Ei["j"];
  D["abij"] -= Ea["a"];
  D["abij"] -= Ea["b"];

  Transform<> div([](double & b){ b=1./b; });
  div(D["abij"]);
  T["abij"] = T["abij"]*D["abij"];
}
```

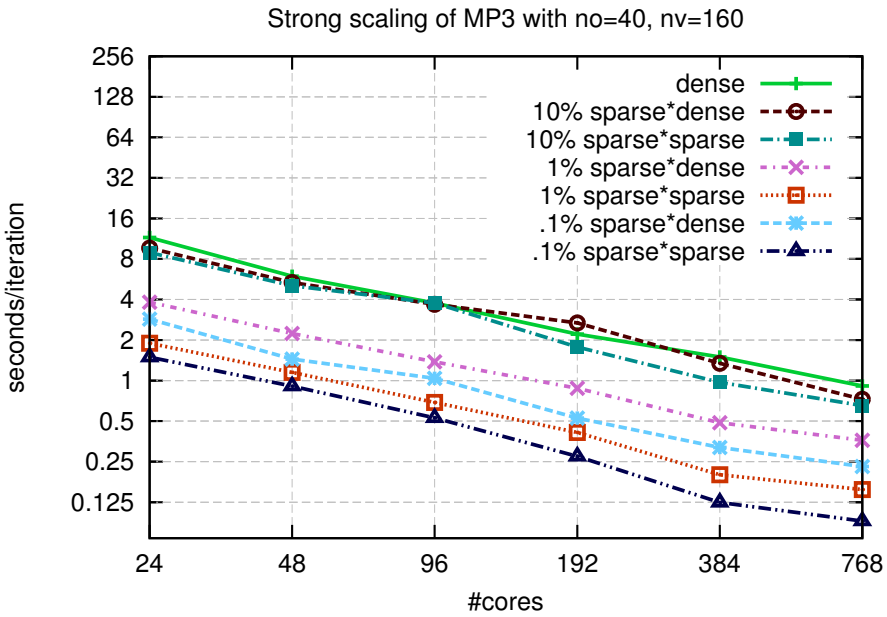
MP3 sparse division

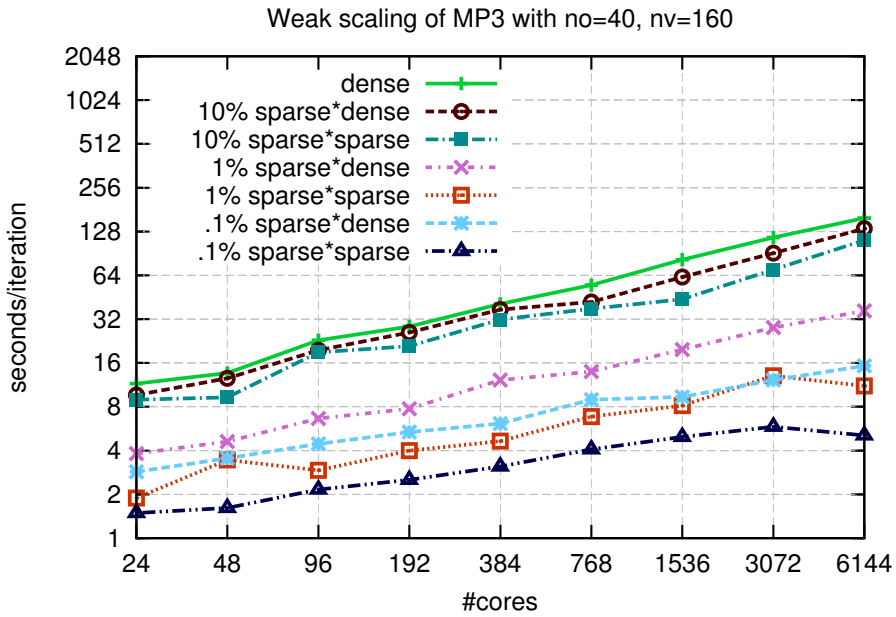
A sparsity-aware version of division in MP2/MP3 using CTF functions

```
struct dp {
  double a, b;
  dp(int x=0){ a=0.0; b=0.0; }
  dp(double a_, double b_){ a=a_, b=b_; }
  dp operator+(dp const & p) const { return dp(a+p.a, b+p.b); }
};

Tensor<dp> TD(4, 1, T.lens, *T.wrld, Monoid<dp,false>());

TD["abij"] = Function<double,dp>(
               [](double d){ return dp(d, 0.0); }
                                )(T["abij"]);

Transform<double,dp> ([](double d, dp & p){ return p.b += d; }
                      )(Ei["i"], TD["abij"]);
... // similar for Ej, Ea, Eb

T["abij"] = Function<dp,double>([](dp p){ return p.a/p.b; }
                                )(TD["abij"]);
```

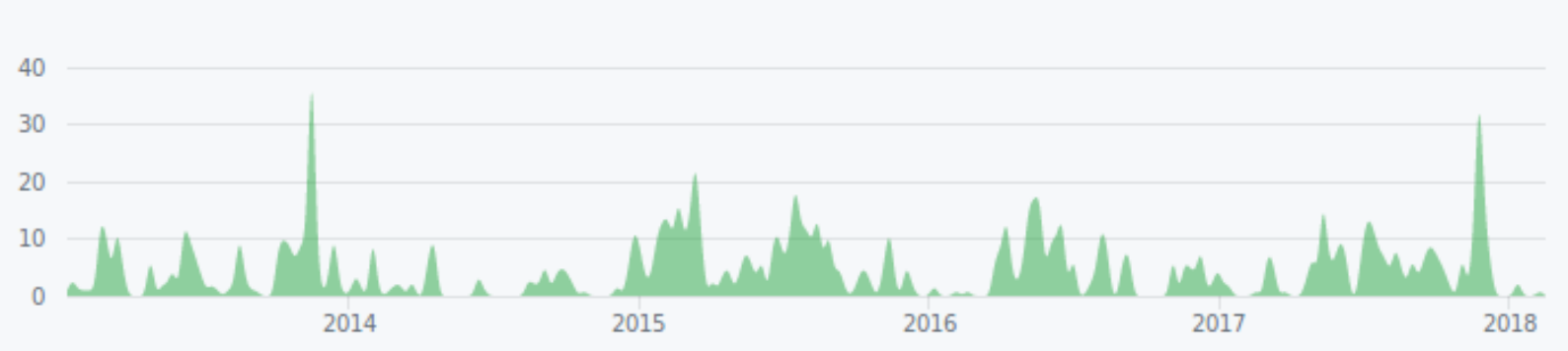
Sparse MP3 code

Strong and weak scaling of sparse MP3 code, with
(1) dense V and T (2) sparse V and dense T (3) sparse V and T





Cyclops release v1.5.1 (January 2018)



- Robust support for Python
  - implementation of `numpy.ndarray` functionality
  - convenient functions for reshape/transposition/slicing (much easier to prototype code than in C++)
  - user-defined elementwise functions not available for direct use
- Integration with other libraries
  - HPTT (Paul Springer), dense transposition ay peak memory bandwidth
  - *batched BLAS*, faster Hadamard product-like contractions, especially with MKL (credit to Eduardo Yap, undergraduate student, UIUC)
  - *ScaLAPACK* conversion mature, SVD available as a simple function in C++/Python (credit to Eric Song, undergraduate student, UIUC)

Future directions and acknowledgements

Future/ongoing directions in Cyclops development

- *General abstractions for tensor decompositions*: HOSVD already available, CP decomposition, tensor train, etc.
- Concurrent scheduling of multiple contractions
- Fourier transforms along tensor modes
- Further Python functionality
- Faster/specialized/optimized tensor slicing

Acknowledgements

- Devin Matthews (UT Austin), Jeff Hammond (Intel Corp.), James Demmel (UC Berkeley), Torsten Hoefler (ETH Zurich), Zecheng Zhang, Eric Song, Eduardo Yap (UIUC)
- Computational resources at NERSC, ALCF, NCSA
- New allocation on XSEDE Stampede 2 for Cyclops on KNL

Backup slides

Performance breakdown on BG/Q

Performance data (from circa 2013) for a CCSD iteration with 200 electrons and 1000 orbitals on 4096 nodes of Mira

4 processes per node, 16 threads per process

Total time: 18 mins

v-orbitals, o-electrons

kernel	% of time	complexity	architectural bounds
DGEMM	45%	$O(v^4o^2/p)$	flops/mem bandwidth
broadcasts	20%	$O(v^4o^2/p\sqrt{M})$	multicast bandwidth
prefix sum	10%	$O(p)$	allreduce bandwidth
data packing	7%	$O(v^2o^2/p)$	integer ops
all-to-all-v	7%	$O(v^2o^2/p)$	bisection bandwidth
tensor folding	4%	$O(v^2o^2/p)$	memory bandwidth

Applications of partially-symmetric tensor contractions

High-accuracy methods in computational quantum chemistry

- solve the multi-electron Schrödinger equation $\boldsymbol{H}|\boldsymbol{\Psi}\rangle = E|\boldsymbol{\Psi}\rangle$, where $\boldsymbol{H}$ is a linear operator, but $\boldsymbol{\Psi}$ is a function of *all* electrons
- use wavefunction ansatze like $\boldsymbol{\Psi} \approx \boldsymbol{\Psi}^{(k)} = e^{\boldsymbol{T}^{(k)}}|\boldsymbol{\Psi}^{(k-1)}\rangle$ where $\boldsymbol{\Psi}^{(0)}$ is a mean-field (averaged) function and $\boldsymbol{T}^{(k)}$ is an order $2k$ tensor, acting as a multilinear excitation operator on the electrons
- coupled-cluster methods use the above ansatze for $k \in \{2, 3, 4\}$ (CCSD, CCSDT, CCSDTQ)
- solve iteratively for $\boldsymbol{T}^{(k)}$, where each iteration has cost $O(n^{2k+2})$, dominated by contractions of partially antisymmetric tensors
- for example, a dominant contraction in CCSD ($k = 2$) is

$$\boldsymbol{Z}_{i\bar{c}}^{a\bar{k}} = \sum_{b=1}^{n}\sum_{j=1}^{n} \boldsymbol{T}_{ij}^{ab} \cdot \boldsymbol{V}_{b\bar{c}}^{j\bar{k}}$$

Our CCSD factorization

Credit to John F. Stanton and Jurgen Gauss

$$
\begin{aligned}
\tau_{ij}^{ab} &= t_{ij}^{ab} + \frac{1}{2} P_b^a P_j^i t_i^a t_j^b, \\
\tilde{F}_e^m &= f_e^m + \sum_{fn} v_{ef}^{mn} t_n^f, \\
\tilde{F}_e^a &= (1-\delta_{ae}) f_e^a - \sum_m \tilde{F}_e^m t_m^a - \frac{1}{2} \sum_{mnf} v_{ef}^{mn} t_{mn}^{af} + \sum_{fn} v_{ef}^{an} t_n^f, \\
\tilde{F}_i^m &= (1-\delta_{mi}) f_i^m + \sum_e \tilde{F}_e^m t_i^e + \frac{1}{2} \sum_{nef} v_{ef}^{mn} t_{in}^{ef} + \sum_{fn} v_{if}^{mn} t_n^f,
\end{aligned}
$$

Our CCSD factorization

$$\tilde{W}_{ei}^{mn} = v_{ei}^{mn} + \sum_f v_{ef}^{mn} t_i^f,$$

$$\tilde{W}_{ij}^{mn} = v_{ij}^{mn} + P_j^i \sum_e v_{ie}^{mn} t_j^e + \frac{1}{2} \sum_{ef} v_{ef}^{mn} \tau_{ij}^{ef},$$

$$\tilde{W}_{ie}^{am} = v_{ie}^{am} - \sum_n \tilde{W}_{ei}^{mn} t_n^a + \sum_f v_{ef}^{ma} t_i^f + \frac{1}{2} \sum_{nf} v_{ef}^{mn} t_{in}^{af},$$

$$\tilde{W}_{ij}^{am} = v_{ij}^{am} + P_j^i \sum_e v_{ie}^{am} t_j^e + \frac{1}{2} \sum_{ef} v_{ef}^{am} \tau_{ij}^{ef},$$

$$z_i^a = f_i^a - \sum_m \tilde{F}_i^m t_m^a + \sum_e f_e^a t_i^e + \sum_{em} v_{ei}^{ma} t_m^e + \sum_{em} v_{im}^{ae} \tilde{F}_e^m + \frac{1}{2} \sum_{efm} v_{ef}^{am} \tau_{im}^{ef}$$

$$- \frac{1}{2} \sum_{emn} \tilde{W}_{ei}^{mn} t_{mn}^{ea},$$

$$z_{ij}^{ab} = v_{ij}^{ab} + P_j^i \sum_e v_{ie}^{ab} t_j^e + P_b^a P_j^i \sum_{me} \tilde{W}_{ie}^{am} t_{mj}^{eb} - P_b^a \sum_m \tilde{W}_{ij}^{am} t_m^b$$

$$+ P_b^a \sum_e \tilde{F}_e^a t_{ij}^{eb} - P_j^i \sum_m \tilde{F}_i^m t_{mj}^{ab} + \frac{1}{2} \sum_{ef} v_{ef}^{ab} \tau_{ij}^{ef} + \frac{1}{2} \sum_{mn} \tilde{W}_{ij}^{mn} \tau_{mn}^{ab},$$