An Overview of Cyclops Tensor Framework

Edgar Solomonik

University of Illinois at Urbana-Champaign

May 8, 2017

A stand-alone library for petascale tensor computations

Cyclops Tensor Framework (CTF)

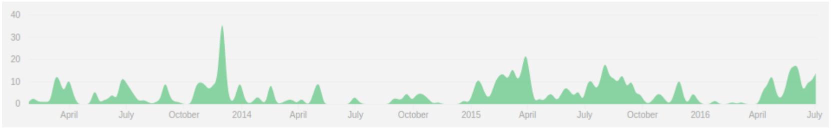
- distributed-memory symmetric/sparse tensors as C++ objects

```
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 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"]);
```

- development (1500 commits) since 2011, open source since 2013

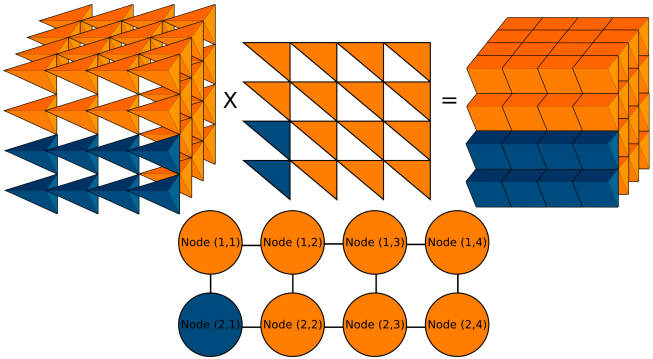


- fundamental part of Aquarius, CC4S, integrated into QChem and Psi4

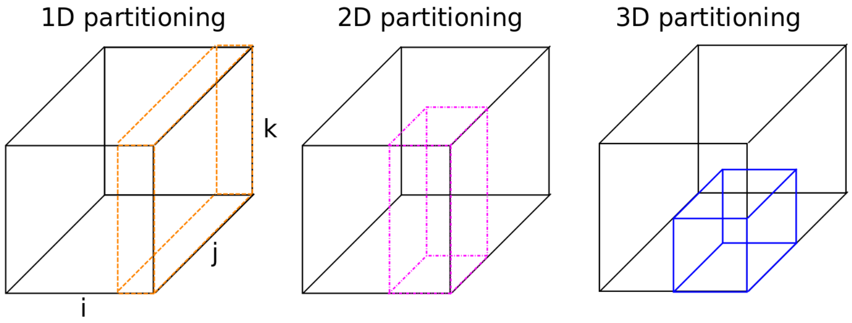
CTF parallel scalability

CTF is tuned for massively-parallel architectures

- multidimensional tensor blocking and processor grids
- cyclic assignment of elements to processors is well-suited for symmetric and sparse tensors
- performance-model-driven decomposition is done at runtime
- optimized redistribution kernels for tensor transposition



Matrix multiplication partitioning



$$C_{ij} = \sum_k A_{ik} B_{kj}$$

Best partitioning depends on dimensions of matrices and number of nonzeros for sparse matrices, tensor contractions are similar

Communication avoiding matrix multiplication

CTF uses the most efficient matrix multiplication algorithms

- the interprocessor communication cost of matrix multiplication $C = AB$ of matrices with dims $m \times k$ and $k \times n$ on p processors is

$$W = \begin{cases} O\left(\min\limits_{p_1p_2p_3=p}\left[\frac{mk}{p_1p_2} + \frac{kn}{p_2p_3} + \frac{mn}{p_1p_3}\right]\right) & \text{: dense} \\ O\left(\min\limits_{p_1p_2p_3=p}\left[\frac{\mathrm{nnz}(A)}{p_1p_2} + \frac{\mathrm{nnz}(B)}{p_2p_3} + \frac{\mathrm{nnz}(C)}{p_1p_3}\right]\right) & \text{: sparse} \end{cases}$$

- communication-optimality depends on memory usage M

$$W = \begin{cases} \Omega\left(\frac{mnk}{p\sqrt{M}}\right) & \text{: dense} \\ \Omega\left(\frac{\mathrm{flops}(A,B,C)}{p\sqrt{M}}\right) & \text{: sparse} \end{cases}$$

- CTF selects best p_1, p_2, p_3 subject to memory usage constraints on M

Data redistribution and matricization

Transitions between contractions require redistribution and refolding

- CTF defines a base distribution for each tensor (by default, over all processors), which can also be user-specified
- before each contraction, the tensor data is redistributed globally and matricized locally
- 3 types of global redistribution algorithms are optimized and threaded
- matricization for sparse tensors corresponds to a conversion to a compressed-sparse-row (CSR) matrix layout
- the cost of redistribution is part of the performance model used to select the contraction algorithm

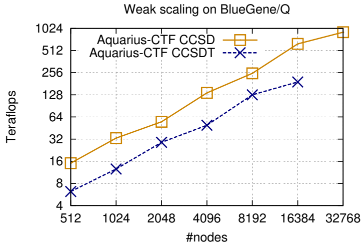
Dense tensor application: coupled cluster using CTF

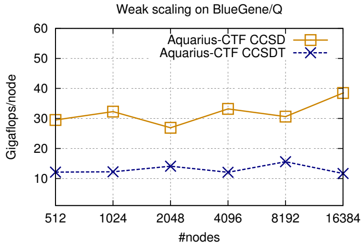
Extracted from Aquarius (lead by Devin Matthews)
https://github.com/devinamatthews/aquarius

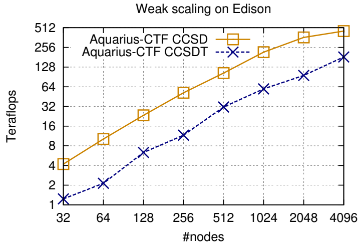
```
FMI["mi"]      += 0.5*WMNEF["mnef"]*T2["efin"];
WMNIJ["mnij"] += 0.5*WMNEF["mnef"]*T2["efij"];
FAE["ae"]      -= 0.5*WMNEF["mnef"]*T2["afmn"];
WAMEI["amei"] -= 0.5*WMNEF["mnef"]*T2["afin"];

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"];
```

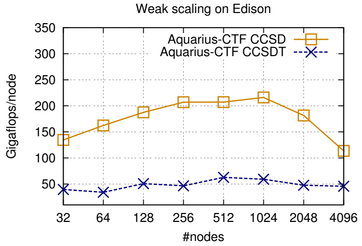
Dense tensor application: coupled cluster performance

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









compares well to NWChem (up to 10x speed-ups for CCSDT)

Sparse tensor application: MP3 calculation

```
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"];
```

A case-study of a naive sparse MP3 code

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"];
}
```

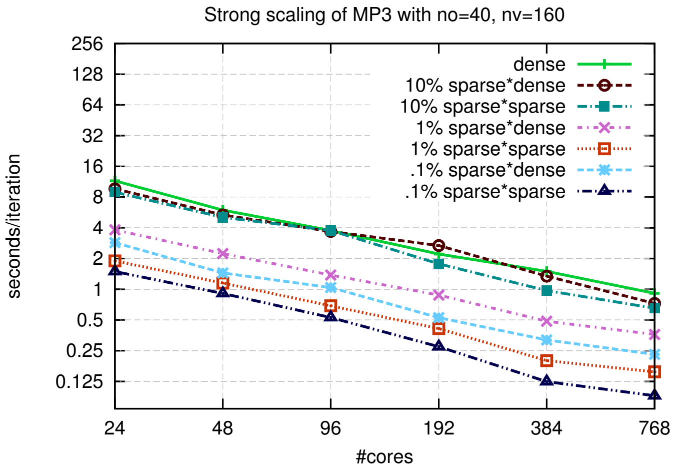
A case-study of a naive sparse MP3 code

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"]);
```

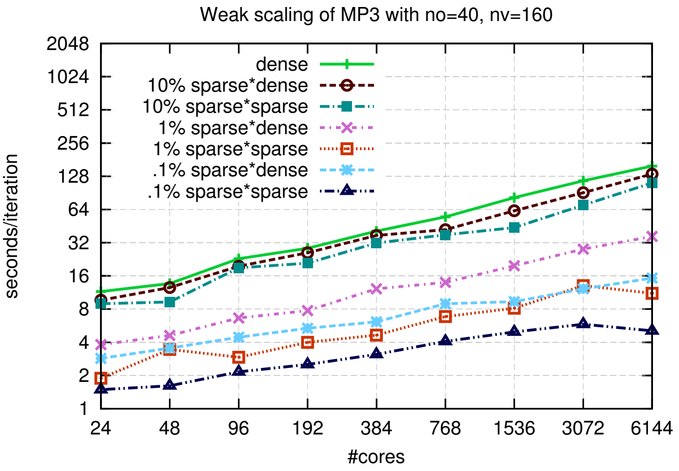
Sparse tensor application: strong scaling

We study the time to solution of the sparse MP3 code, with
(1) dense V and T **(2)** sparse V and dense T **(3)** sparse V and T



Sparse tensor application: weak scaling

We study the scaling to larger problems of the sparse MP3 code, with **(1)** dense V and T **(2)** sparse V and dense T **(3)** sparse V and T



Interoperability

A Python interface for CTF is currently in development

- Cython is used to expose C++ routines to Python
- interoperability/back-end for numpy
- `numpy.einsum` and array slicing implemented

Conversions to/from ScaLAPACK have been recently added

- selected ScaLAPACK matrix factorization routines likely to be interfaced in the future

CTF status and explorations

Much ongoing work and future directions in CTF for quantum chemistry

- performance improvement for batched tensor operations
- predefined output sparsity for contractions
- abstractions for tensor factorizations

Also lots of applications beyond quantum chemistry

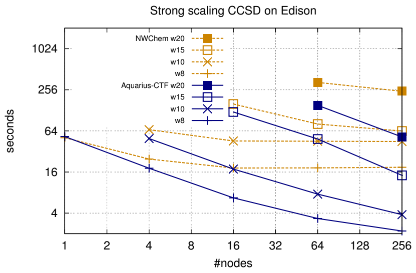
- lattice QCD
- algebraic multigrid
- finite and spectral element methods
- shortest path computation in graphs and betweenness centrality
- FFT, bitonic sort, parallel scan, HSS matrix computations
- convolutional neural networks

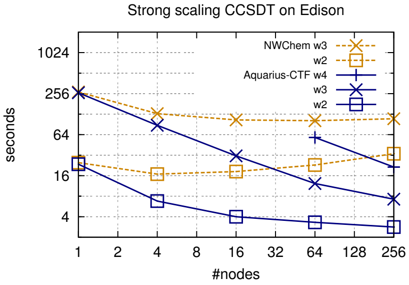
Backup slides

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



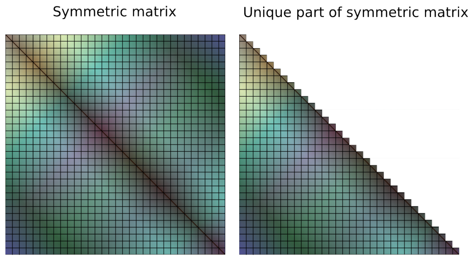


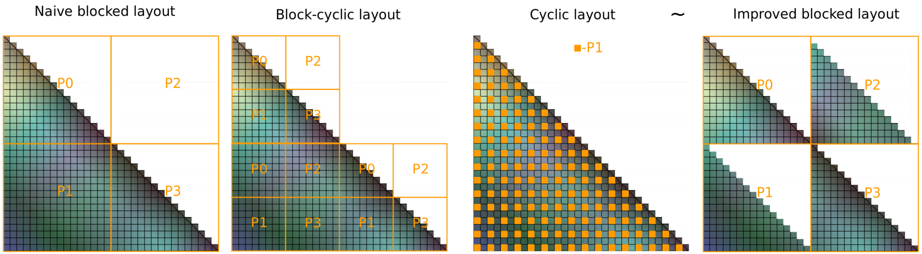
How does CTF achieve parallel scalability?

CTF algorithms address fundamental parallelization challenges:

- load balance
- communication costs
  - amount of data sent or received
  - number of messages sent or received
  - amount of data moved between memory and cache
  - ~~amount of data moved between memory and disk~~

Balancing load via a cyclic data decomposition



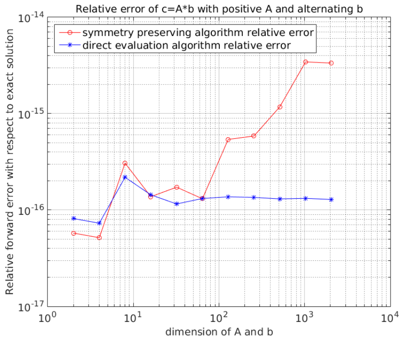


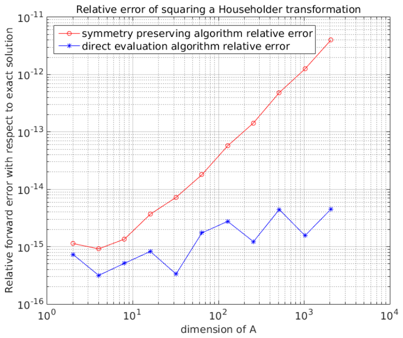
for sparse tensors, a cyclic layout also provides a load-balanced distribution

Our CCSD factorization

$$
\begin{aligned}
\tilde{W}^{mn}_{ei} &= v^{mn}_{ei} + \sum_f v^{mn}_{ef} t^f_i, \\
\tilde{W}^{mn}_{ij} &= v^{mn}_{ij} + P^i_j \sum_e v^{mn}_{ie} t^e_j + \frac{1}{2}\sum_{ef} v^{mn}_{ef} \tau^{ef}_{ij}, \\
\tilde{W}^{am}_{ie} &= v^{am}_{ie} - \sum_n \tilde{W}^{mn}_{ei} t^a_n + \sum_f v^{ma}_{ef} t^f_i + \frac{1}{2}\sum_{nf} v^{mn}_{ef} t^{af}_{in}, \\
\tilde{W}^{am}_{ij} &= v^{am}_{ij} + P^i_j \sum_e v^{am}_{ie} t^e_j + \frac{1}{2}\sum_{ef} v^{am}_{ef} \tau^{ef}_{ij}, \\
z^a_i &= f^a_i - \sum_m \tilde{F}^m_i t^a_m + \sum_e f^a_e t^e_i + \sum_{em} v^{ma}_{ei} t^e_m + \sum_{em} v^{ae}_{im} \tilde{F}^m_e + \frac{1}{2}\sum_{efm} v^{am}_{ef} \tau^{ef}_{im} \\
&- \frac{1}{2}\sum_{emn} \tilde{W}^{mn}_{ei} t^{ea}_{mn}, \\
z^{ab}_{ij} &= v^{ab}_{ij} + P^i_j \sum_e v^{ab}_{ie} t^e_j + P^a_b P^i_j \sum_{me} \tilde{W}^{am}_{ie} t^{eb}_{mj} - P^a_b \sum_m \tilde{W}^{am}_{ij} t^b_m \\
&+ P^a_b \sum_e \tilde{F}^a_e t^{eb}_{ij} - P^i_j \sum_m \tilde{F}^m_i t^{ab}_{mj} + \frac{1}{2}\sum_{ef} v^{ab}_{ef} \tau^{ef}_{ij} + \frac{1}{2}\sum_{mn} \tilde{W}^{mn}_{ij} \tau^{ab}_{mn},
\end{aligned}
$$

Stability of symmetry preserving algorithms





Performance breakdown on BG/Q

Performance data 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