A Distributed Memory Library for Sparse Tensor Functions and Contractions

Edgar Solomonik

University of Illinois at Urbana-Champaign

February 27, 2017

Tensor abstractions for parallel computing

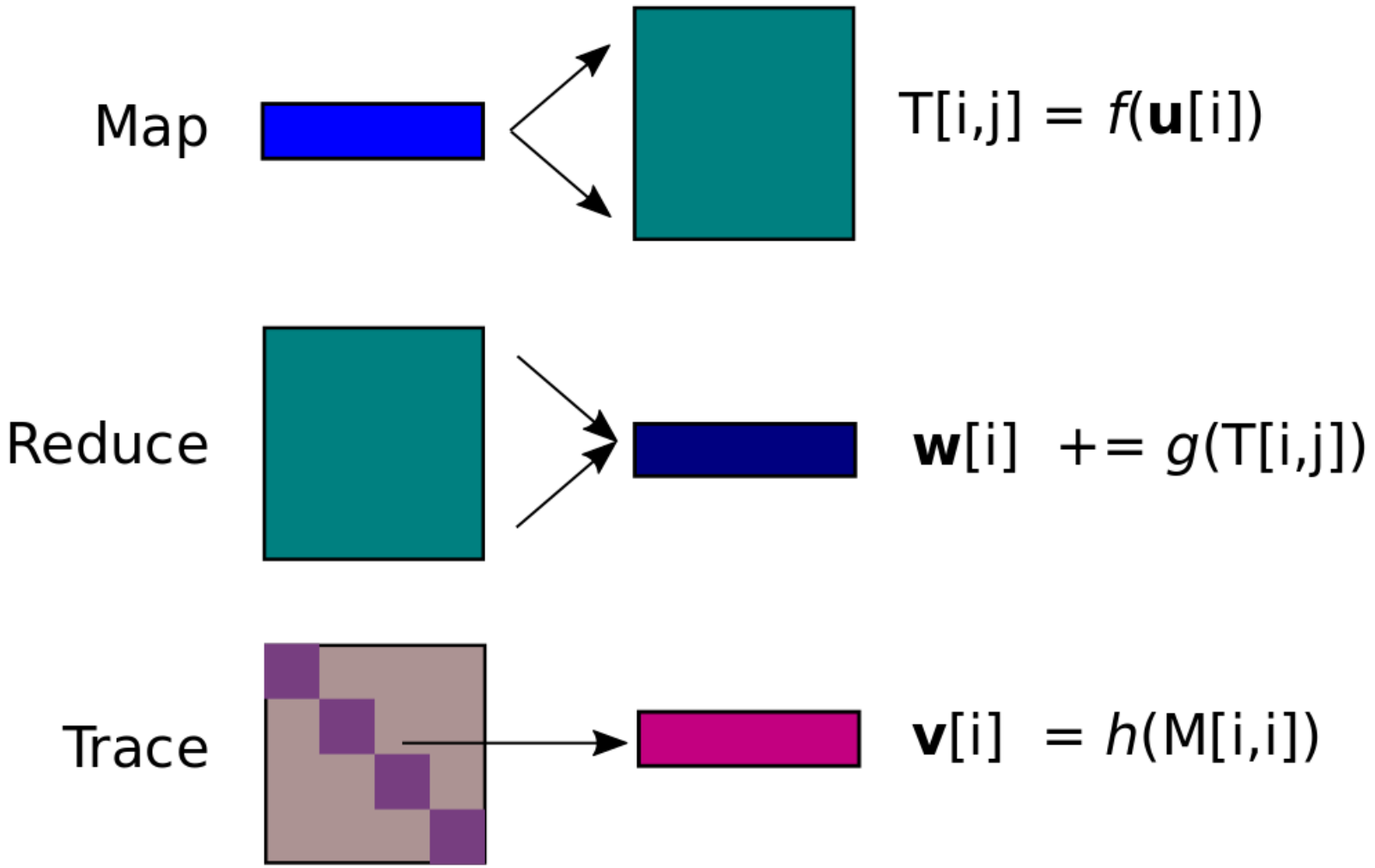
Algebraic tensor operations are a natural language for massive datasets

- tensors are multidimensional arrays with attributes
  - sparsity
    $$M_{ij} \neq 0 \quad \text{if} \quad (i,j) \in S$$
  - symmetry
    $$M_{ij} = M_{ji} \quad \text{or} \quad M_{ij} = -M_{ji}$$
  - algebraic structure
    $$M_{ij} + M_{kl} =? \quad \text{and} \quad M_{ij} \cdot M_{kl} =?$$

- bulk-synchronous tensor operations
  - tensor summation/contraction define high-level data transformations
  - induced from scalar operations (algebraic structure of element function)
- plus everything we want from multidimensional arrays (slicing, etc.)

Generalized tensor summation

A mapping $\mathbb{R}^{d_1 \times \cdots \times d_n} \rightarrow \mathbb{R}^{d_1 \times \cdots \times d_m}$ induced by element operations



Generalized tensor contraction

A mapping $\mathbb{R}^{d_1 \times \cdots \times d_n} \times \mathbb{R}^{d_k \times \cdots \times d_{n+l}} \rightarrow \mathbb{R}^{d_1 \times \cdots \times d_{k+s} \times d_{n+1} \times \cdots \times d_{n+l}}$

- $s = 0$ defines a single tensor contraction
  - dot product
  - matrix-vector multiplication
  - matrix-matrix multiplication
  - tensor-times-matrix
- $s > 0$ defines many independent tensor contractions
  - pointwise vector product
  - Hadamard matrix product
  - batched matrix multiplication

Applications of high-order tensor representations

Numerical solution to differential equations

- spectral element methods
- higher-order differential operators

Computer vision and graphics

- 2D image $\otimes$ angle $\otimes$ time
- classification, compression (tensor factorizations, sparsity)

Machine learning

- convolutional neural networks, high-order statistics
- reduced-order models, recommendation systems (tensor factorizations)

Graph computations

- hypergraphs, time-dependent graphs
- clustering/partitioning/path-finding (eigenvector computations)

Divide-and-conquer algorithms representable by tensor folding

- bitonic sort, FFT, scans, HSS matrix-vector multiplication

Tensors for computational chemistry/physics

Manybody Schrödinger equation

- "curse of dimensionality" – exponential state space

Condensed matter physics

- tensor network models (e.g. DMRG), tensor per lattice site
- highly symmetric multilinear tensor representation
- exponential state space localized $\rightarrow$ factorized tensor form

Quantum chemistry (electronic structure calculations)

- models of molecular structure and chemical reactions
- methods for calculating electronic correlation:
  - "Post Hartree-Fock": configuration interaction, coupled cluster, Møller-Plesset perturbation theory
- multi-electron states as tensors,
  e.g. electron $\otimes$ electron $\otimes$ orbital $\otimes$ orbital
- nonlinear equations of partially (anti)symmetric tensors
- interactions diminish with distance $\rightarrow$ sparsity, low rank

A stand-alone library for petascale tensor computations

Cyclops Tensor Framework (CTF)

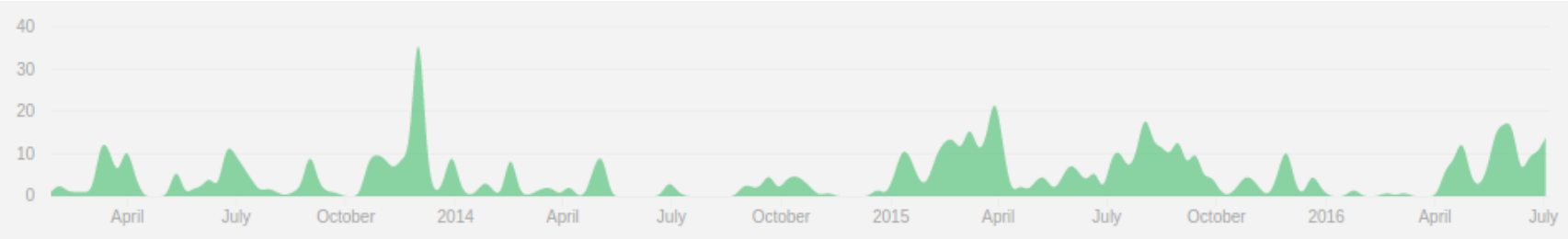
- distributed-memory symmetric/sparse tensors as C++ objects

```
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

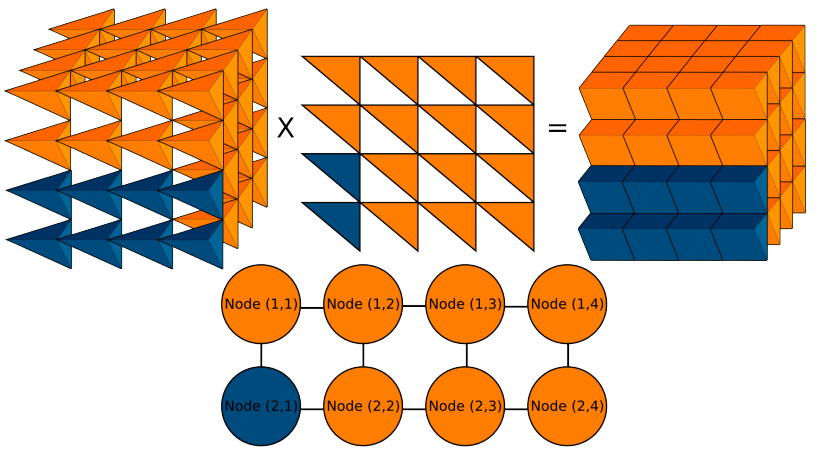


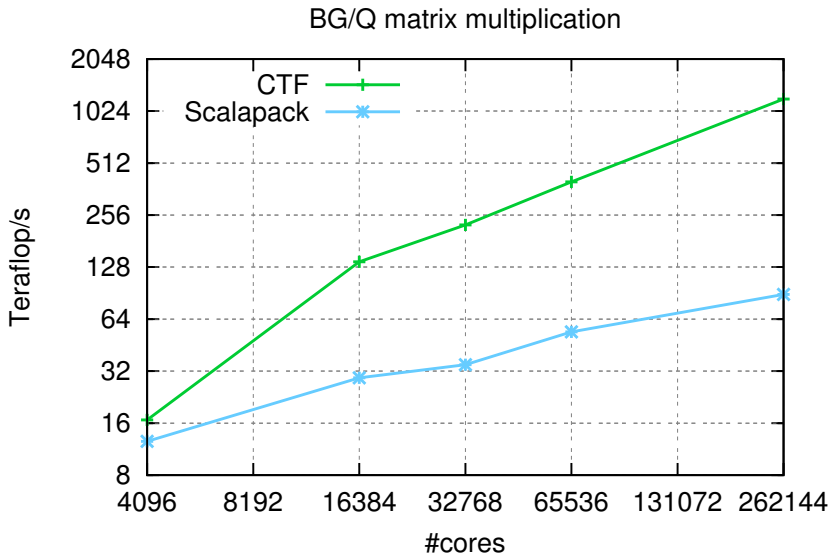
- NEW: Python! towards autoparallel numpy ndarray: einsum, slicing, etc.

CTF parallel scalability

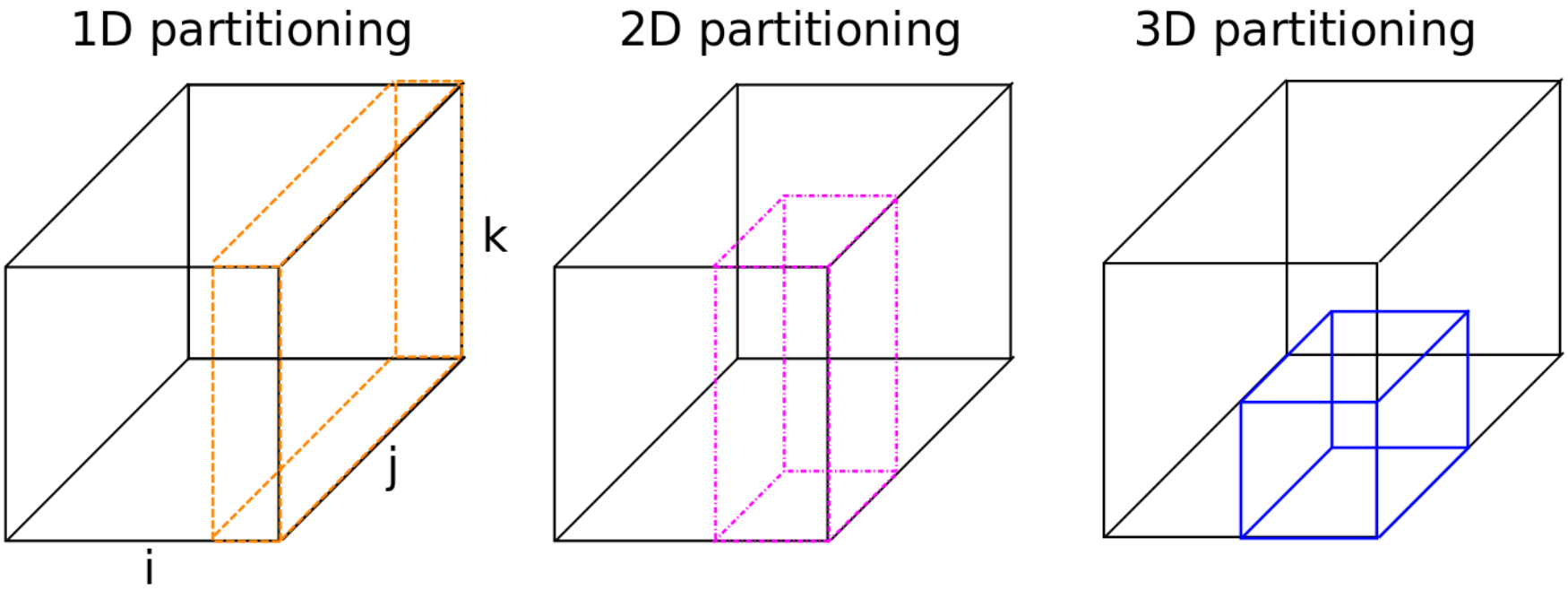
CTF is tuned for massively-parallel architectures

- multidimensional tensor blocking and processor grids
- topology-aware mapping and collective communication
- performance-model-driven decomposition 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

Communication avoiding matrix multiplication

CTF uses the most efficient matrix multiplication algorithms

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

$$W = \begin{cases} O\left(\min_{p_1 p_2 p_3 = p} \left[\frac{mk}{p_1 p_2} + \frac{kn}{p_2 p_3} + \frac{mn}{p_1 p_3} \right] \right) & \text{: dense} \\ O\left(\min_{p_1 p_2 p_3 = p} \left[\frac{\text{nnz}(A)}{p_1 p_2} + \frac{\text{nnz}(B)}{p_2 p_3} + \frac{\text{nnz}(C)}{p_1 p_3} \right] \right) & \text{: sparse} \end{cases}$$

- communication-optimality depends on memory usage M

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

- 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

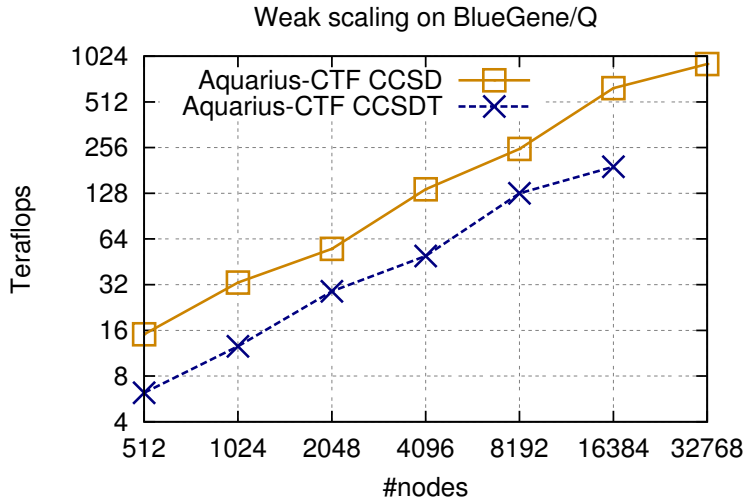
Extracted from Aquarius (lead by Devin Matthews)
https://github.com/devinamatthews/aquarius

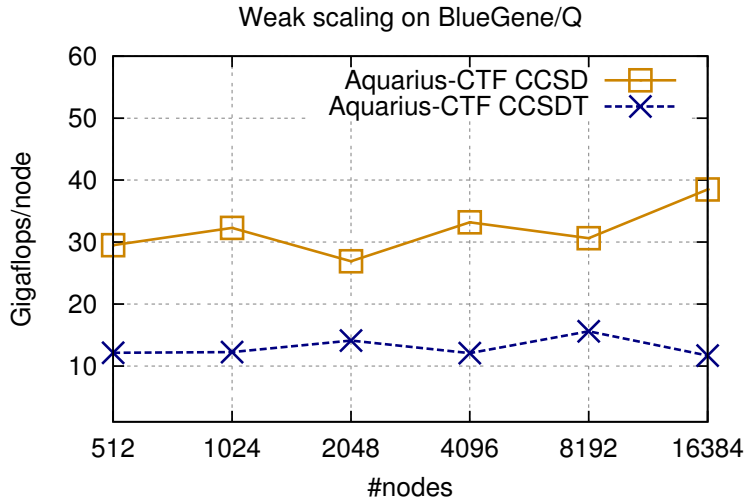
```
FMI["mi"]     += 0.5*WMNEF["mnef"]*T2["efin"];
WMNIJ["mnij"] += 0.5*WMNEF["mnef"]*T2["efij"];
FAE["ae"]     -= 0.5*WMNEF["mnef"]*T2["afmn"];
WAMEI["amei"] -= 0.5*WMNEF["mnef"]*T2["afin"];

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"];
```

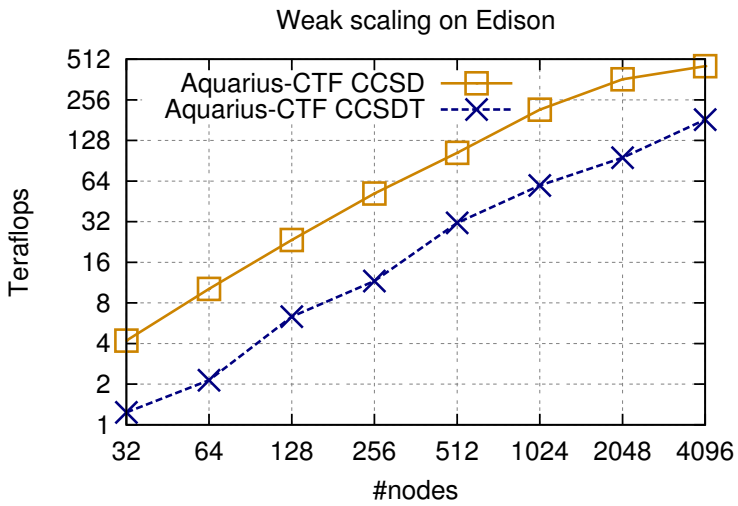
Dense tensor application: coupled cluster performance

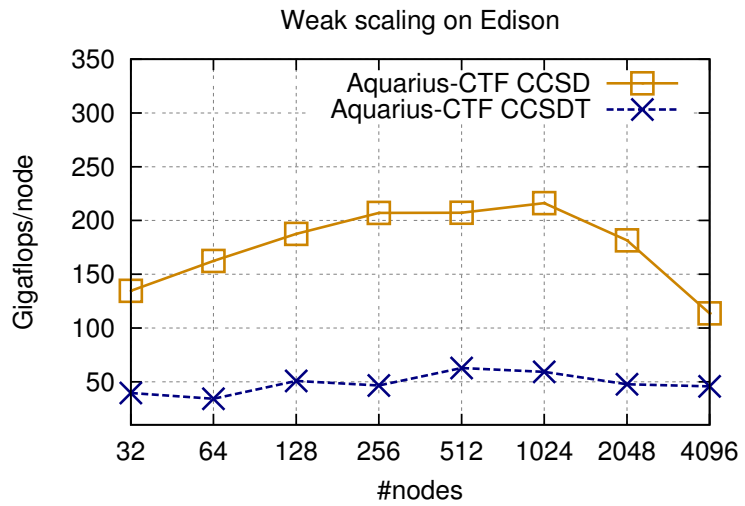
CCSD up to 55 (50) water molecules with cc-pVDZ

CCSDT up to 10 water molecules with cc-pVDZ









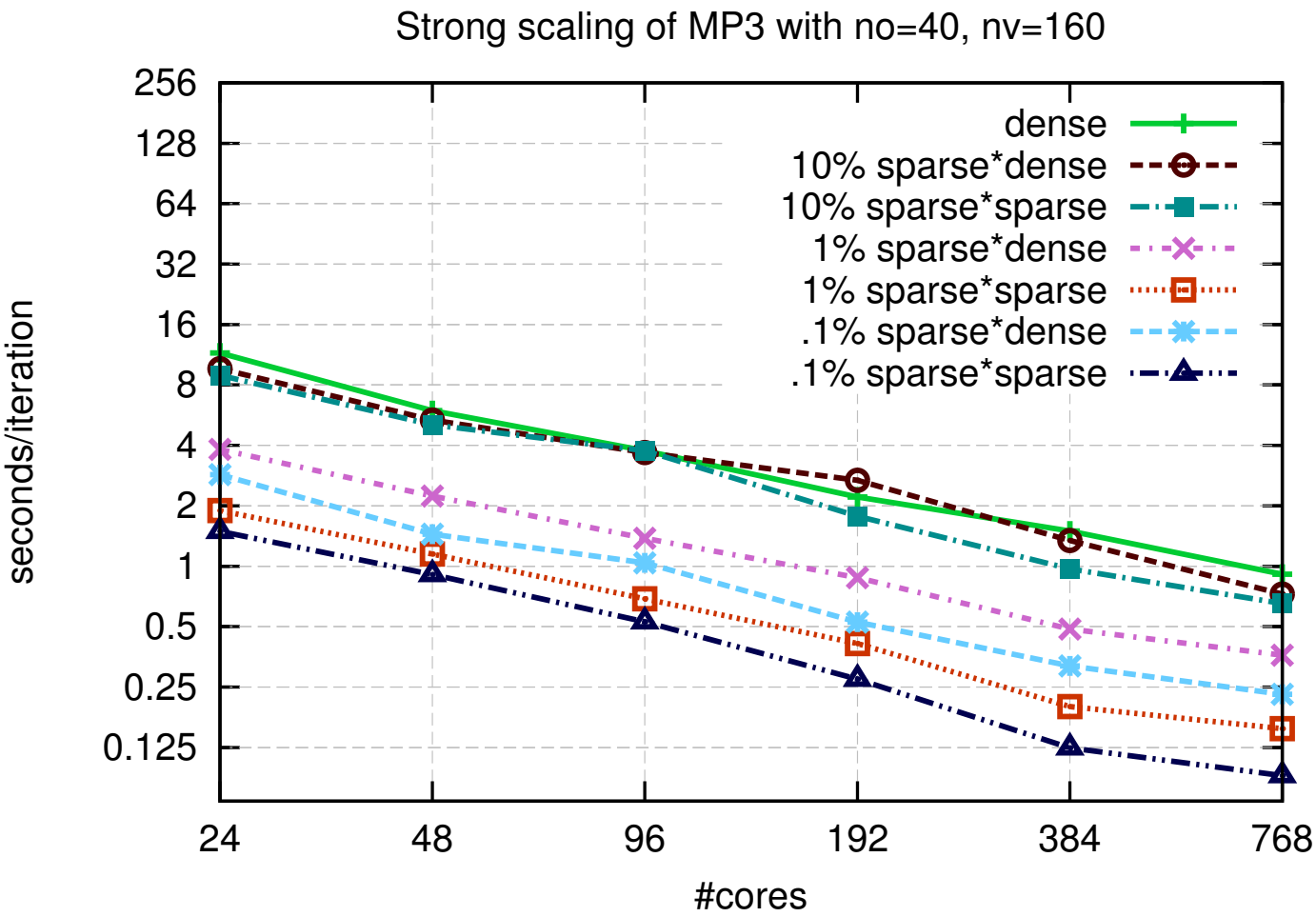
compares well to NWChem (up to 10x speed-ups for CCSDT)

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"];
```

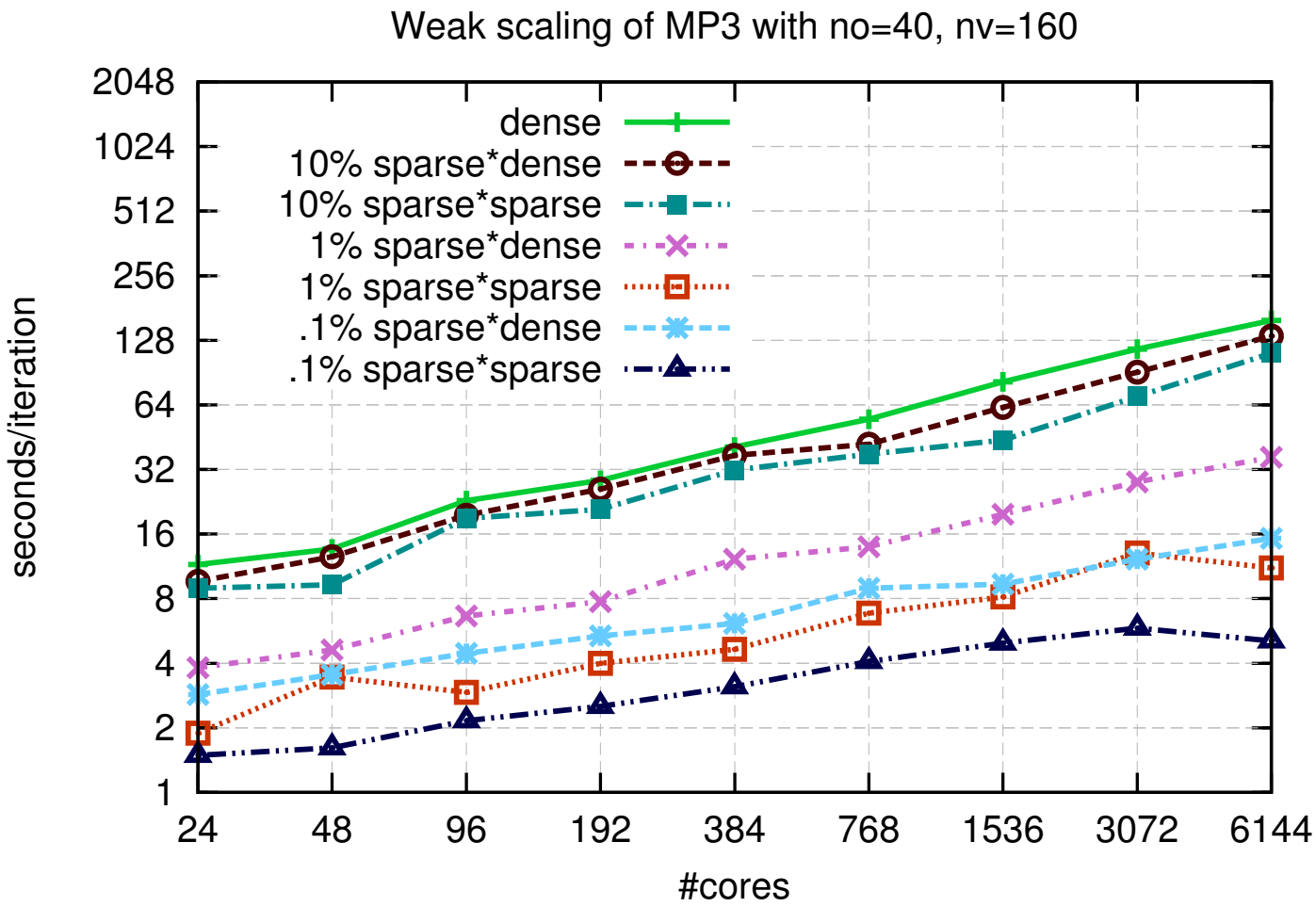
Sparse tensor application: strong scaling

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



Special operator application: betweenness centrality

Betweenness centrality is the importance of vertices in a shortest path tree

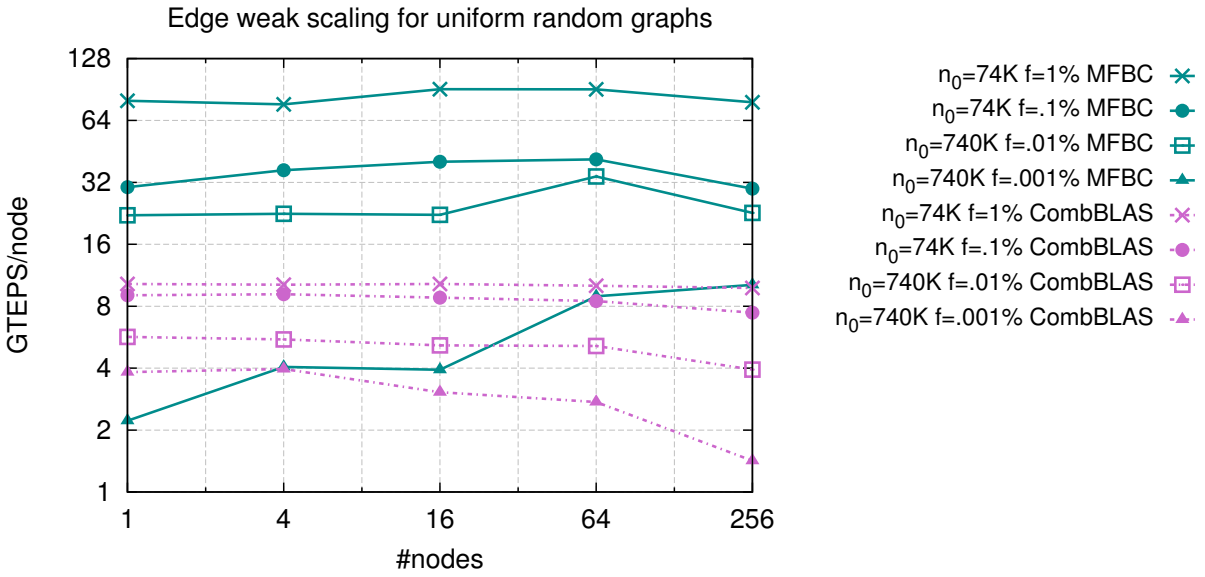
- can be computed via all-pairs shortest-path from distance matrix, but possible to do via less memory (Brandes' algorithm)
- unweighted graphs
  - Breadth First Search (BFS) for each vertex
  - back-propagation of centrality scores along BFS tree
- weighted graphs
  - SSSP for each vertex (we use Bellman Ford with sparse frontiers)
  - back-propagation of betweenness centrality scores (no harder than unweighted)
- our formulation uses a set of starting vertices (many BFS runs), cas as sparse matrix times sparse matrix

Special operator application: betweenness centrality

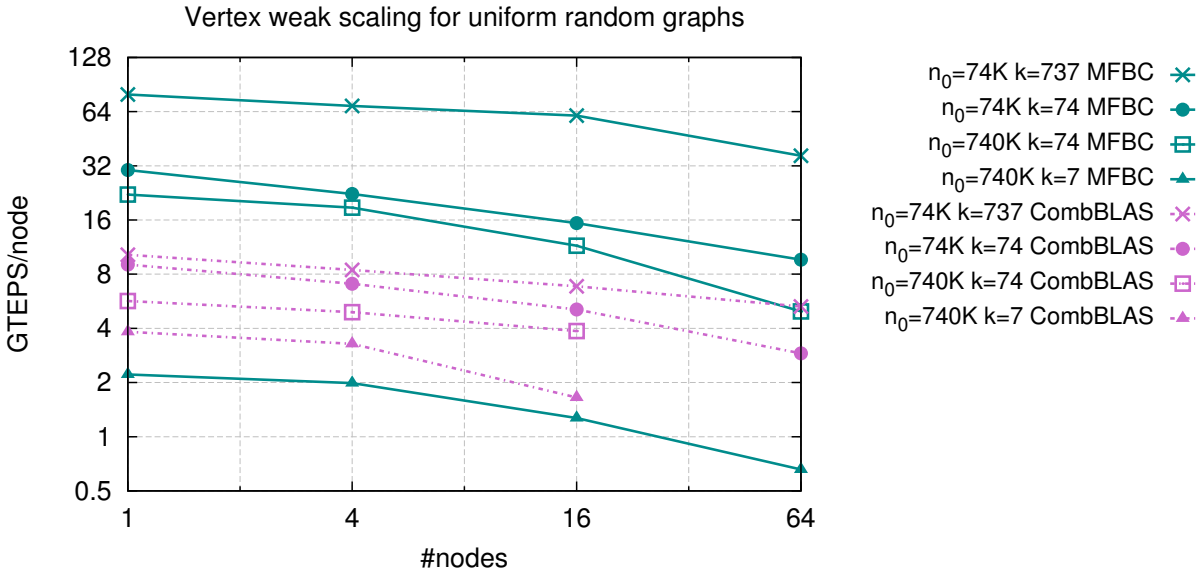
Betweenness centrality code snippet, for k of n nodes

```
void btwn_central(Matrix<int> A, Matrix<path> P, int n, int k){
  Monoid<path> mon(...,
                   [](path a, path b){
                     if (a.w<b.w) return a;
                     else if (b.w<a.w) return b;
                     else return path(a.w, a.m+b.m);
                   }, ...);

  Matrix<path> Q(n,k,mon); // shortest path matrix
  Q["ij"] = P["ij"];

  Function<int,path> append([](int w, path p){
                       return path(w+p.w, p.m);
                     }; );

  for (int i=0; i<n; i++)
    Q["ij"] = append(A["ik"],Q["kj"]);
  ...
}
```

Special operator application: weak scaling from n_0 vertices

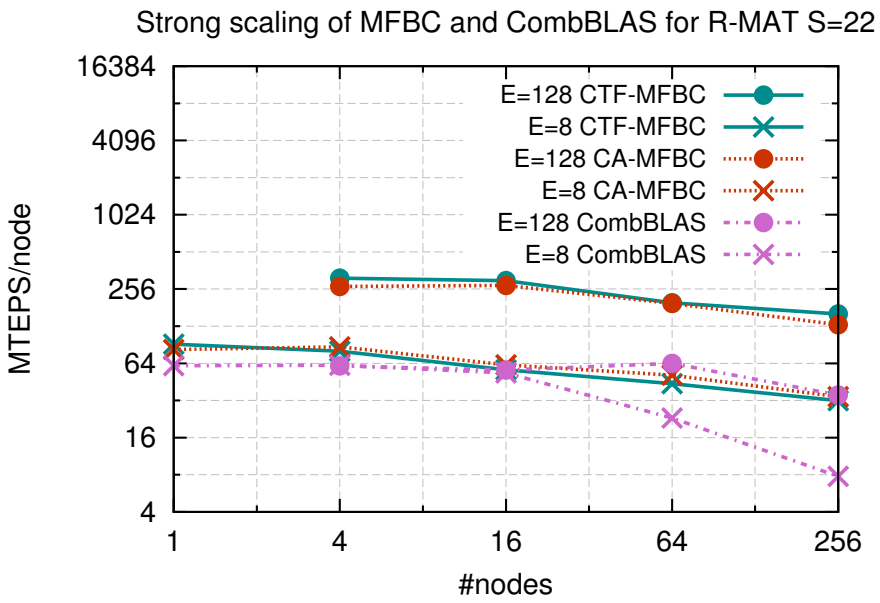


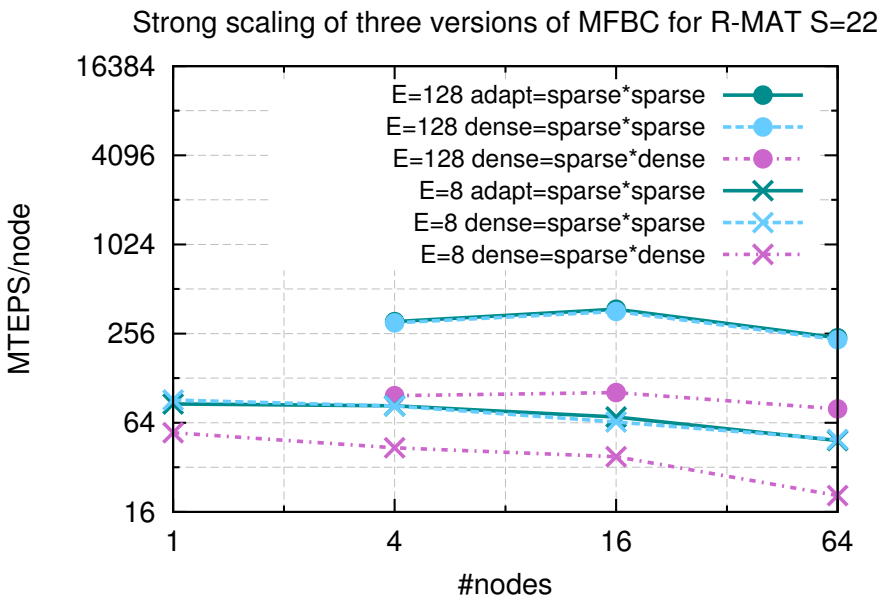
f – fraction of nonzeros



k – vertex degree

Special operator application: strong scaling





Two comparisons of CTF (MFBC) performance

- first comparison of different algorithms
  - with CombBLAS
  - with CA-MFBC (statically-mapped comm-efficient matrix distribution)
- second comparison of different matrix represenations
  - adjacency matrix sparse for all versions
  - frontier sparse or dense rectangular matrix
  - vertices adjacent to frontier (output) sparse or dense rectangular matrix

CTF status and explorations

Much ongoing work and future directions in CTF

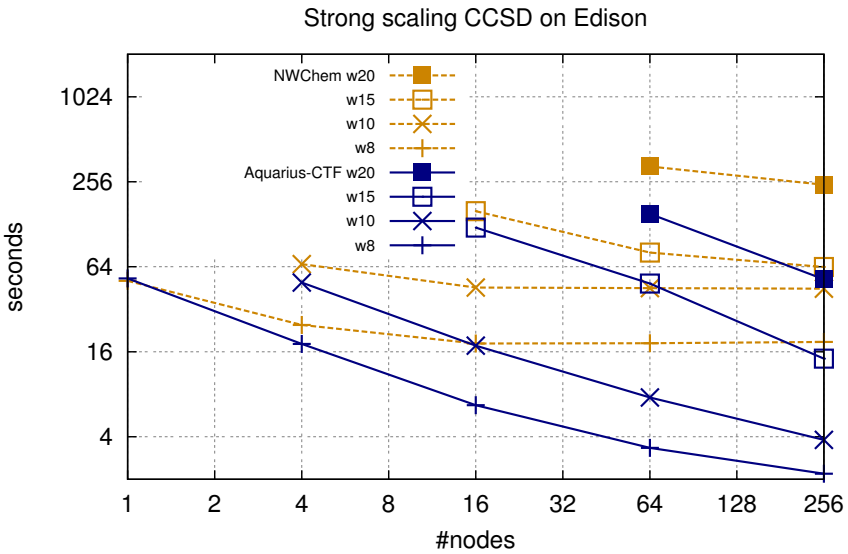
- other existing applications
  - algebraic multigrid: easy implementation, but performance not competitive for SpMV and multigrid restriction matrices with special structure
  - spectral element methods: easy implementation, performance not competitive (CTF Hadamard products are currently slow)
  - FFT, bitonic sort, parallel scan (future: HSS matrix computations)
- ongoing and future work
  - active: development of Python interface (einsum and slicing work)
  - recent: hook-ups for conversion to ScaLAPACK format
  - future: performance improvement for batched tensor operations
  - future: predefined output sparsity for contractions
  - future: tensor factorizations
- existing collaborations and external applications
  - Aquarius (lead by Devin Matthews)
  - QChem via Libtensor (integration lead by Evgeny Epifanovsky)
  - QBall (DFT code, just matrix multiplication)
  - CC4S (lead by Andreas Grüneis)
  - early collaborations involving Lattice QCD and DMRG

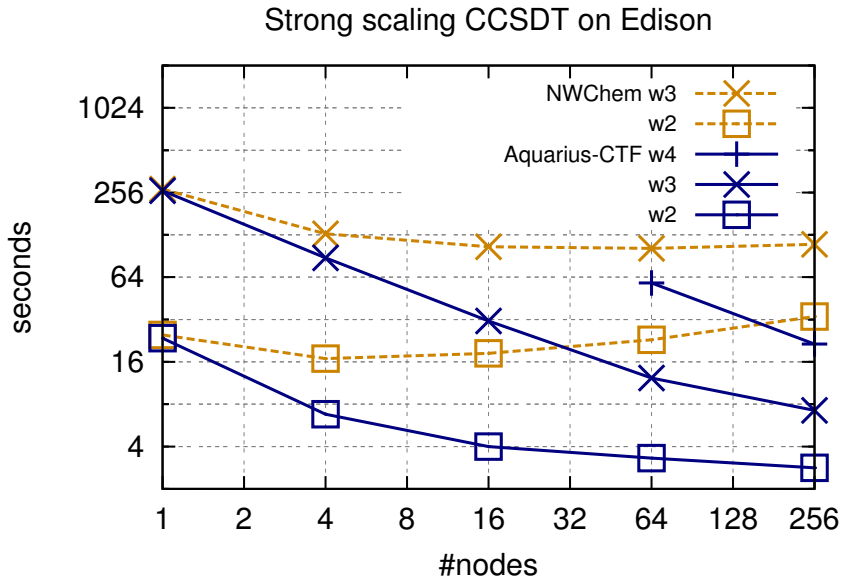
Backup slides

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



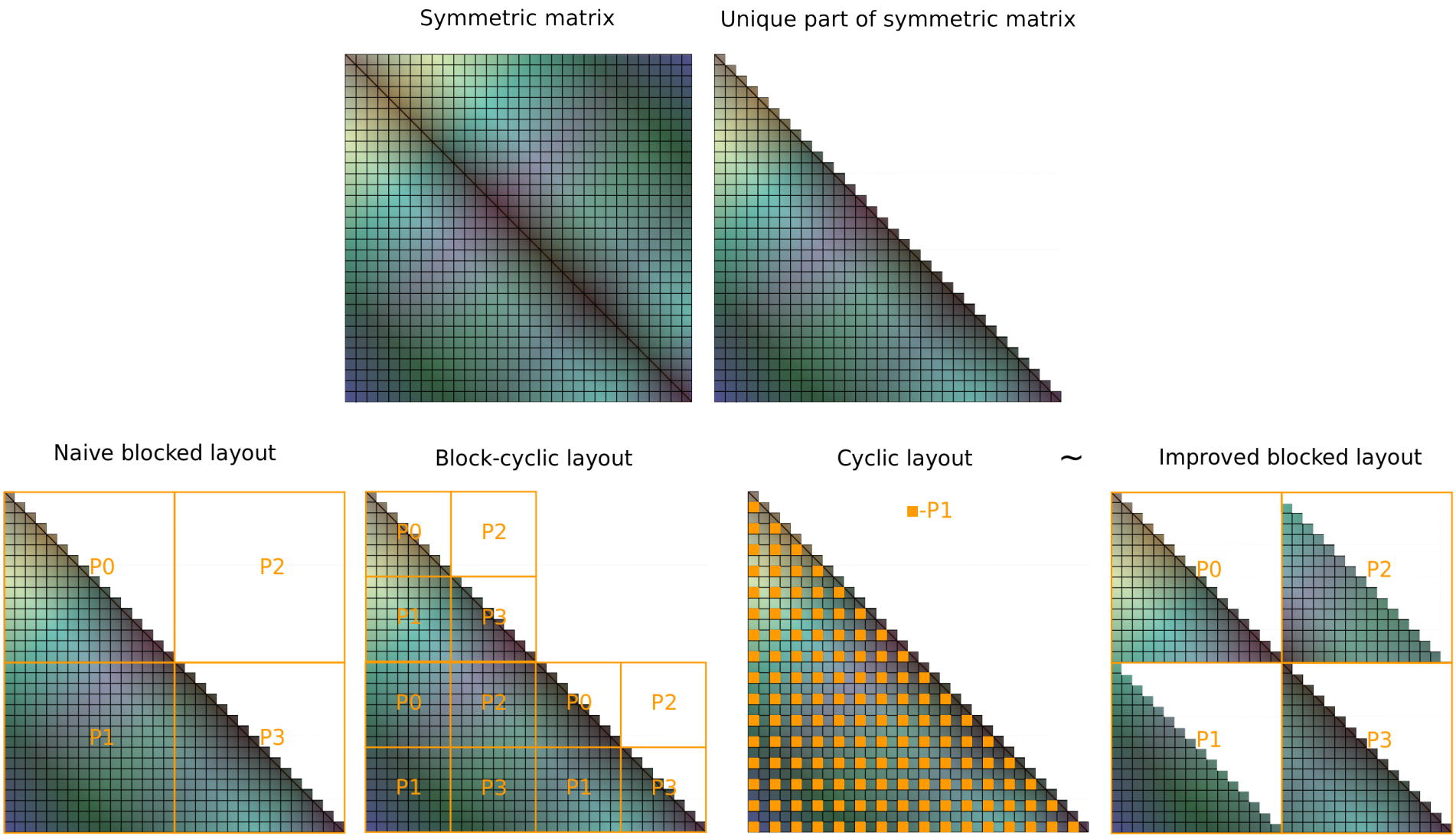


How does CTF achieve parallel scalability?

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

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

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

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

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

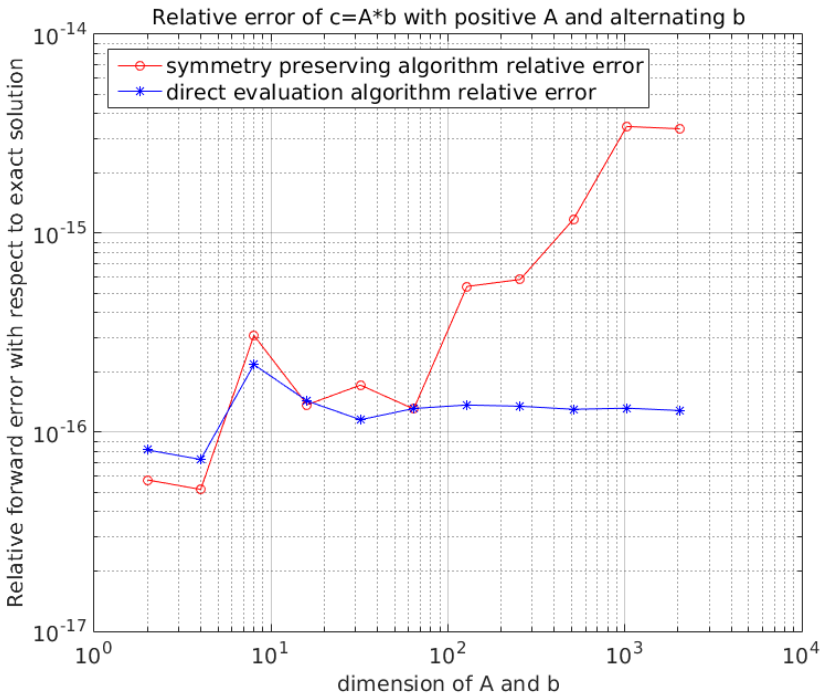
$$z_i^a = f_i^a - \sum_m \tilde{F}_i^m t_m^a + \sum_e f_e^a t_i^e + \sum_{em} v_{ei}^{ma} t_m^e + \sum_{em} v_{im}^{ae} \tilde{F}_e^m + \frac{1}{2} \sum_{efm} v_{ef}^{am} \tau_{im}^{ef}$$

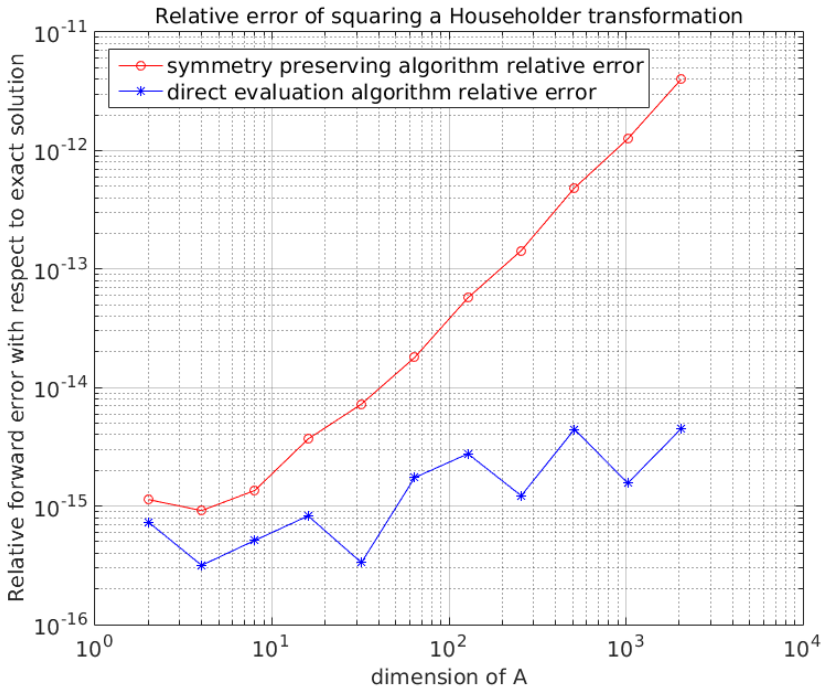
$$- \frac{1}{2} \sum_{emn} \tilde{W}_{ei}^{mn} t_{mn}^{ea},$$

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

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

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