Scalable Tensor Algorithms for Scientific Computing

Edgar Solomonik

Department of Computer Science
University of Illinois at Urbana-Champaign

PMAA 2018

June 28, 2018

A library for parallel tensor computations

Cyclops Tensor Framework (CTF)[1]

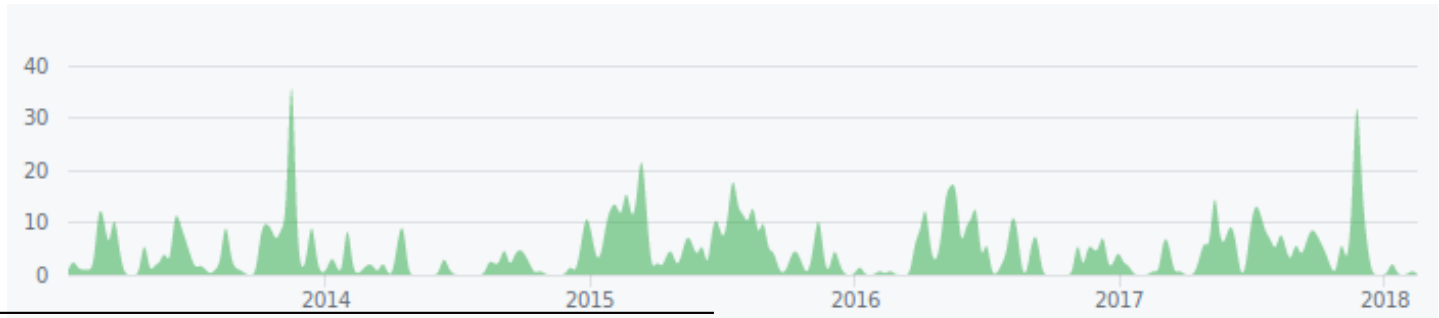
- distributed-memory symmetric/sparse/dense tensor objects

```
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"]);
```

- ∼2000 commits since 2011, open source since 2013



[1] E.S., D. Matthews, J.R. Hammond, J. Demmel, JPDC 2014

Electronic structure calculations with cyclops

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

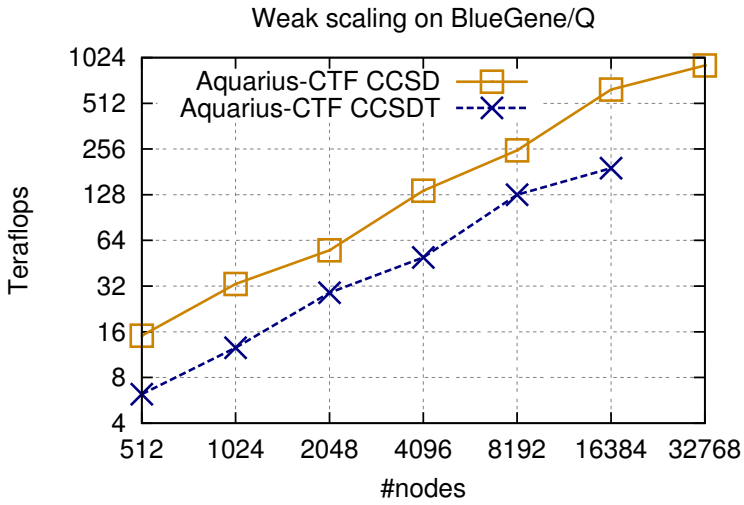
```
FMI["mi"]     += 0.5*WMNEF["mnef"]*T2["efin"];
WMNIJ["mnij"] += 0.5*WMNEF["mnef"]*T2["efij"];
FAE["ae"]     -= 0.5*WMNEF["mnef"]*T2["afmn"];
WAMEI["amei"] -= 0.5*WMNEF["mnef"]*T2["afin"];

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"];
```

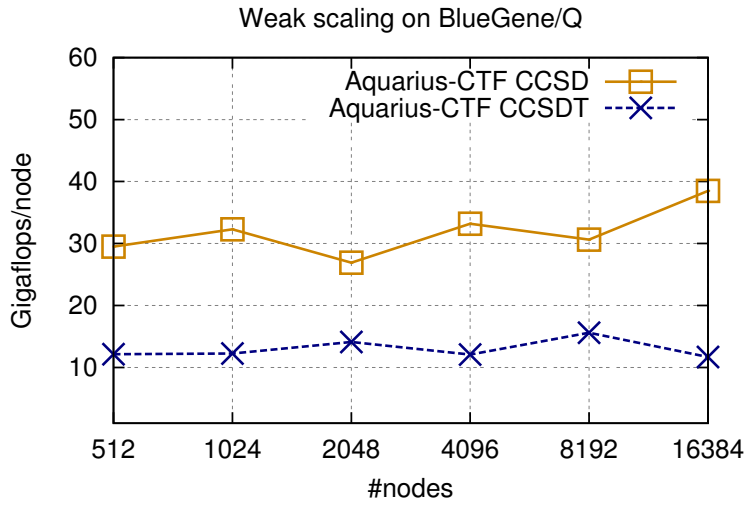
- CTF has been integrated with **QChem, VASP, and Psi4**
- Is also being used for other applications, e.g. by IBM+LLNL collaboration to perform 49-qubit quantum circuit simulation

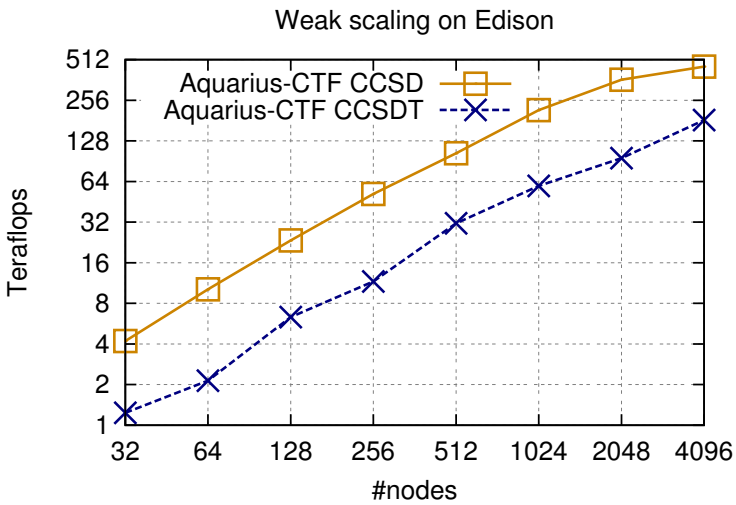
Electronic structure calculations with Cyclops

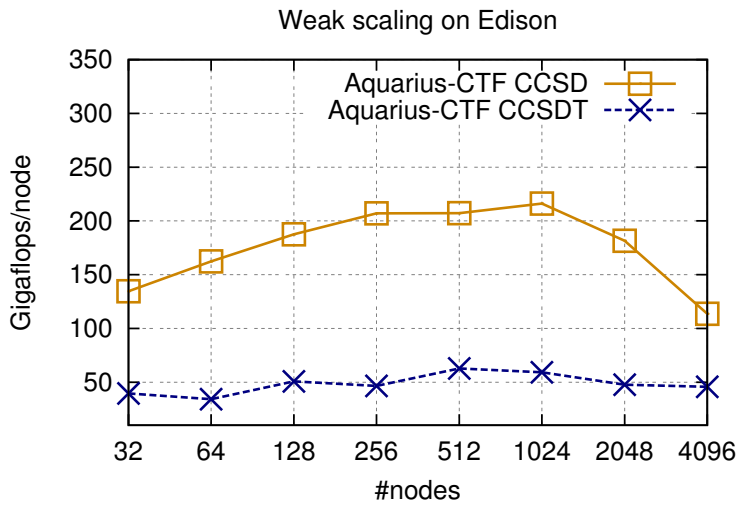
CCSD up to 55 (50) water molecules with cc-pVDZ

CCSDT up to 10 water molecules with cc-pVDZ









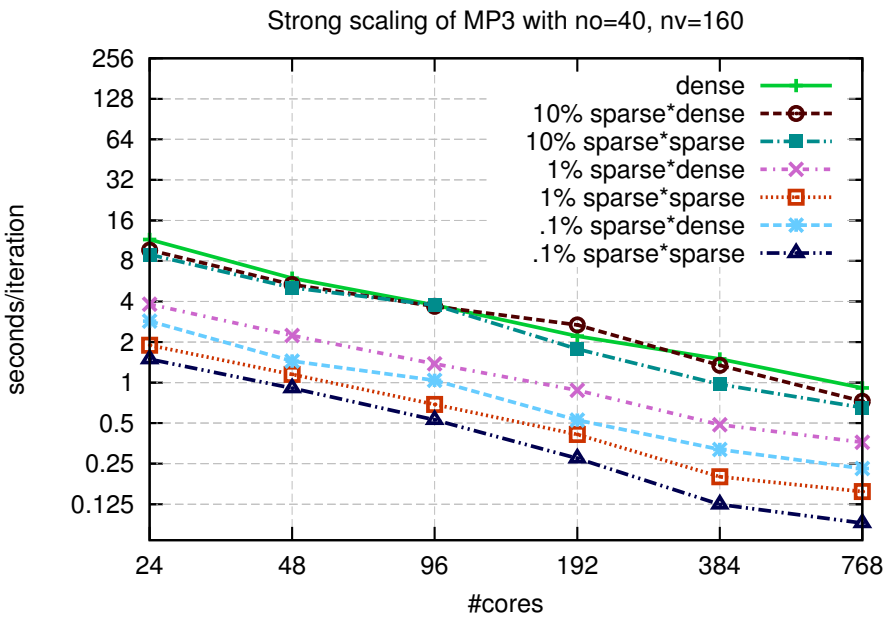
compares well to NWChem (up to 10x speed-ups for CCSDT)

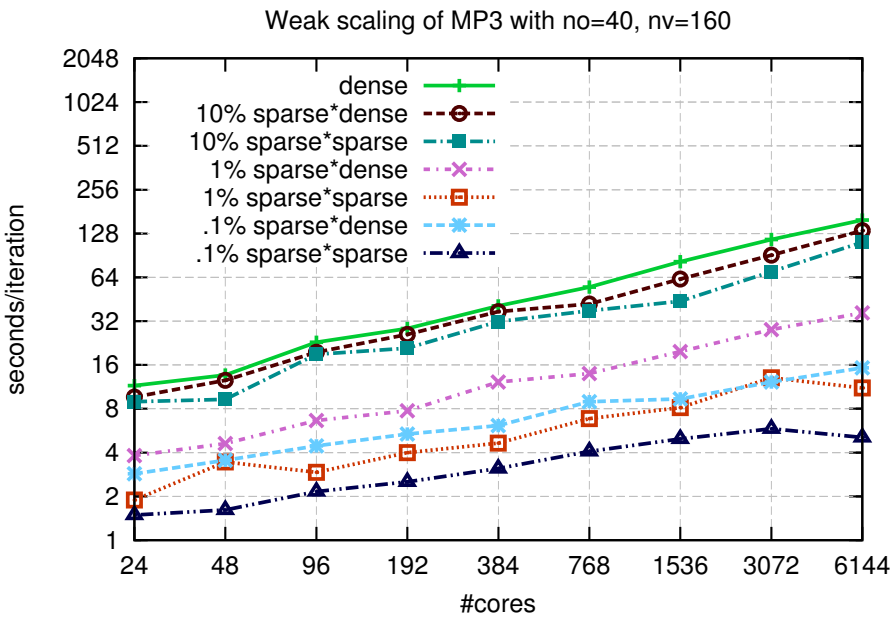
MP3 method

```
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"];
```

Sparse MP3 code

Strong and weak scaling of sparse MP3 code, with
(1) dense $\boldsymbol{V}$ and $\boldsymbol{T}$ (2) sparse $\boldsymbol{V}$ and dense $\boldsymbol{T}$ (3) sparse $\boldsymbol{V}$ and $\boldsymbol{T}$





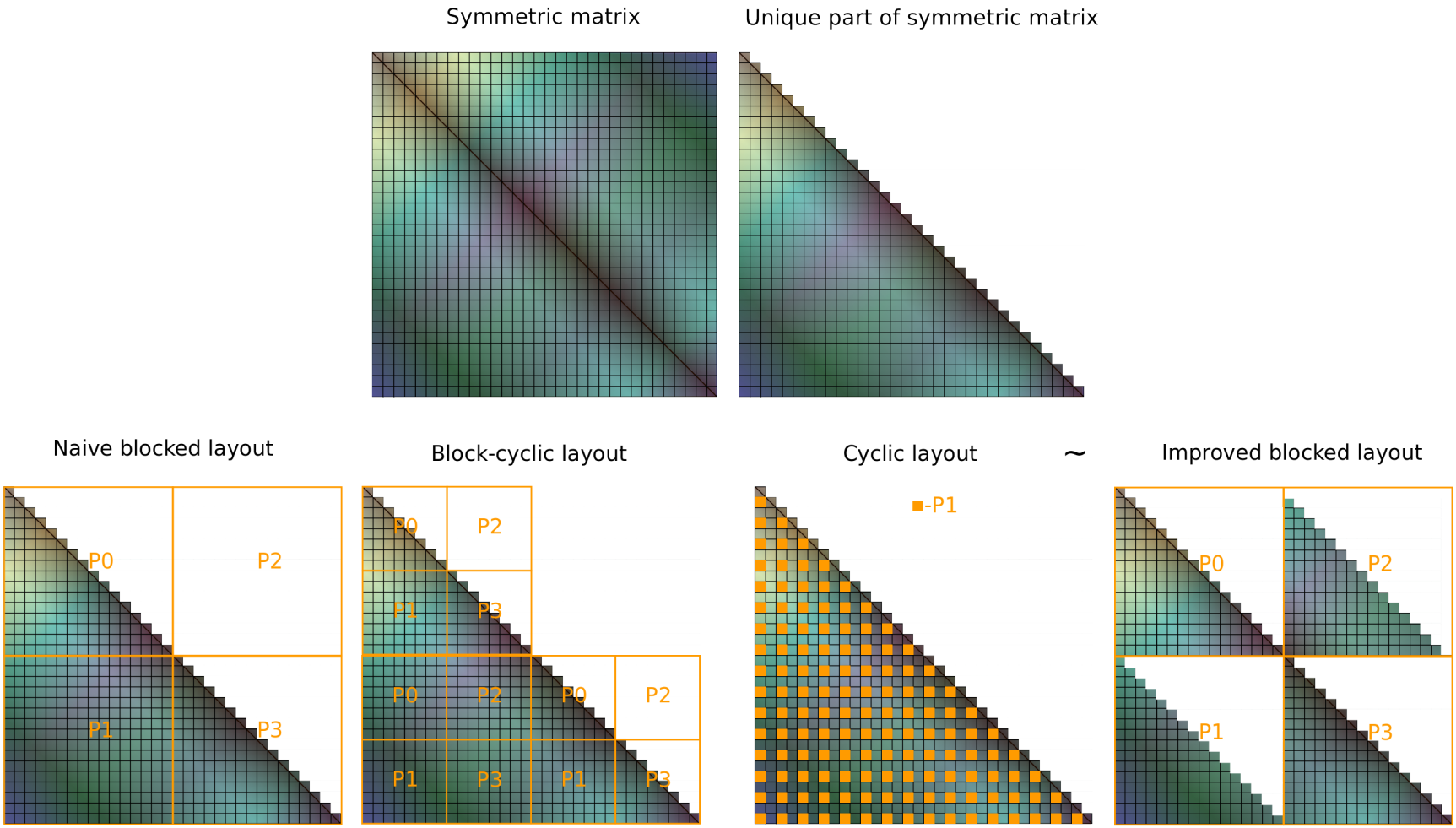
Custom tensor element types

Cyclops permits arbitrary element types and custom functions

- CombBLAS/GraphBLAS-like functionality
- See examples for SSSP, APSP, betweenness centrality, MIS, MIS-2
- Functionality to handle serialization of pointers within user-defined types is under development
- Can already do block-sparsity via sparse tensor (local) of dense tensors (parallel)

```
Matrix< Matrix<> > C(nblk, nblk, SP, self_world, tmon);

C["ij"] = Function< Matrix<> >(
            [](Matrix<> mA, Matrix<> mB){
              mC["ij"] += mA["ik"]*mB["kj"];
              return mC;
            }
          )(A["ik"],B["kj"]);
```

Symmetry and sparsity by cyclicity



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

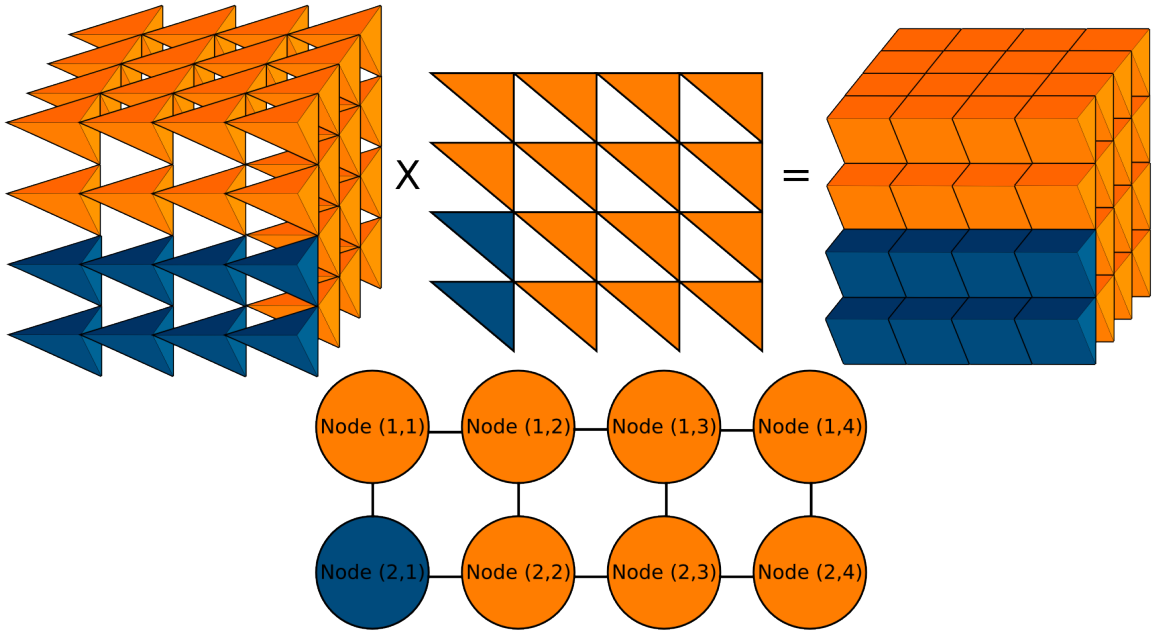
Parallel contraction in Cyclops

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, naturally generalized to sparse matrix products

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 each 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, including batched GEMM
- 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 coefficients $\boldsymbol{x}^*$
- 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 $\boldsymbol{t}$

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

Cyclops with Python

- Using `Cython`, we have provided a Python interface for Cyclops
- Follows `numpy.ndarray` conventions, plus sparsity and MPI execution

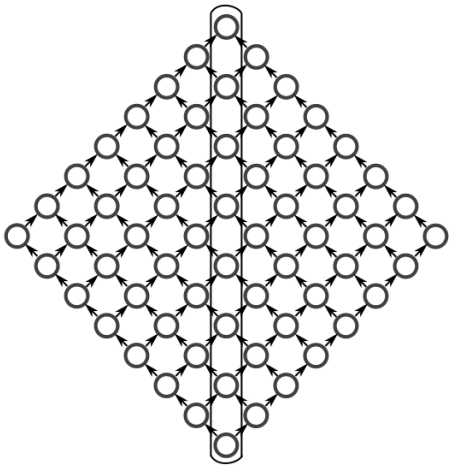
```
Z["abij"]   += V["ijab"];                    // C++
Z.i("abij") << V.i("ijab")                   // Python
W["mnij"]   += 0.5*W["mnef"]*T["efij"];      // C++
W.i("mnij") << 0.5*W.i("mnef")*T.i("efij") // Python
einsum("mnef,efij->mnij",W,T)  // numpy-style Python
```

- Python interface is under active development, but is functional and available (DEMO)

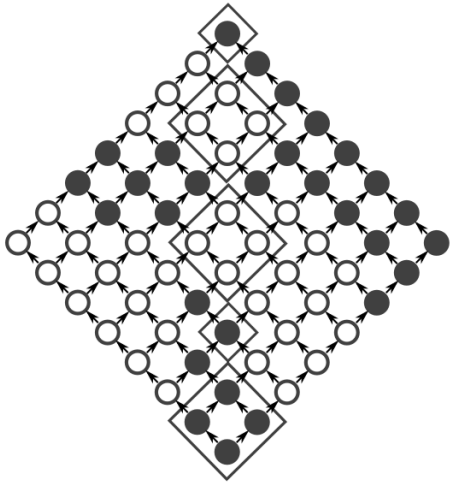
Tradeoffs in the diamond DAG

Computation vs synchronization tradeoff for the $n \times n$ diamond DAG,[1]

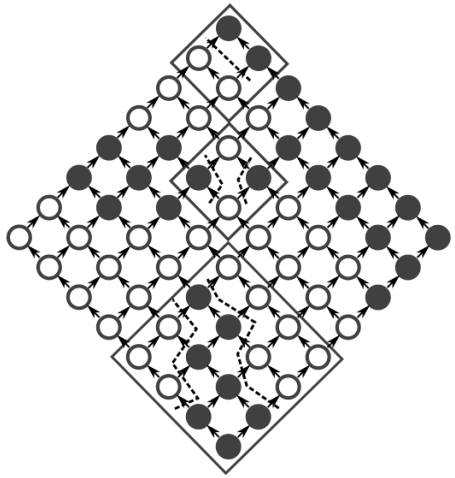
$$F \cdot S = \Omega(n^2)$$



Dependency chain P



Monochrome dependency intervals



Multicolored dependency intervals

In this DAG, vertices denote scalar computations in an algorithm

[1] C.H. Papadimitriou, J.D. Ullman, SIAM JC, 1987

Scheduling tradeoffs of path-expander graphs

Definition ((ϵ, σ)-**path-expander**)

Graph $G = (V, E)$ is a (ϵ, σ)-**path-expander** if there exists a path $(u_1, \ldots u_n) \subset V$, such that the dependency interval $[u_i, u_{i+b}]_G$ for each i, b has size $\Theta(\sigma(b))$ and a minimum cut of size $\Omega(\epsilon(b))$.

Theorem (Path-expander communication lower bound)

Any parallel schedule of an algorithm with a (ϵ, σ)-**path-expander** *dependency graph about a path of length* n *and some* $b \in [1, n]$ *incurs computation* (F)*, communication* (W)*, and synchronization* (S) *costs:*

$$F = \Omega\left(\sigma(b) \cdot n/b\right), \quad W = \Omega\left(\epsilon(b) \cdot n/b\right), \quad S = \Omega\left(n/b\right).$$

Corollary

If $\sigma(b) = b^d$ *and* $\epsilon(b) = b^{d-1}$*, the above theorem yields,*

$$F \cdot S^{d-1} = \Omega(n^d), \quad W \cdot S^{d-2} = \Omega(n^{d-1}).$$

3D algorithms for matrix computations

For Cholesky factorization with p processors, cost is

$$F = \Theta(n^3/p), \quad W = \Theta(n^2/p^\delta), \quad S = \Theta(p^\delta)$$

for any $\delta = [1/2, 2/3]$.
Achieving similar costs for LU, QR, and the symmetric eigenvalue problem requires algorithmic changes.

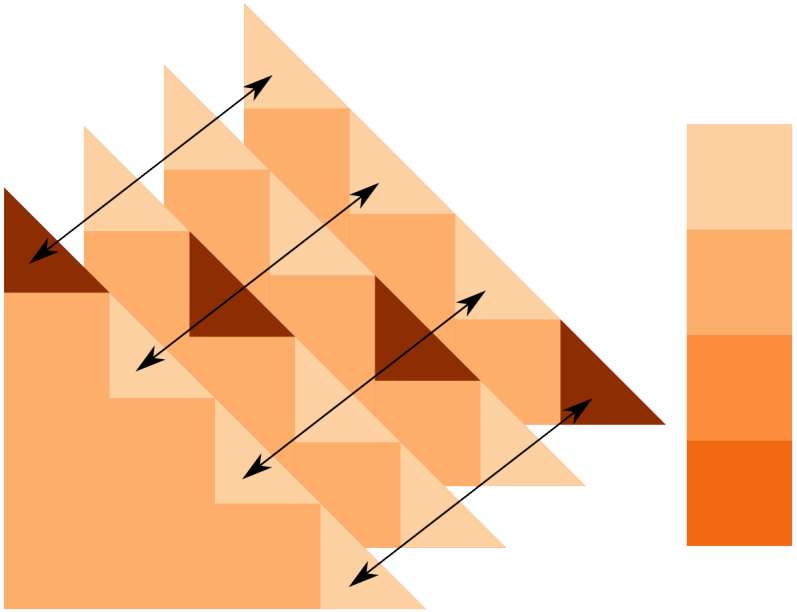
triangular solve	square TRSM ✓[1]	rectangular TRSM ✓[2]
LU with pivoting	pairwise pivoting ✓[3]	tournament pivoting ✓[4]
QR factorization	Givens on square ✓[3]	Householder on rect. ✓[5]
SVD	singular values only ✓[5]	singular vectors X

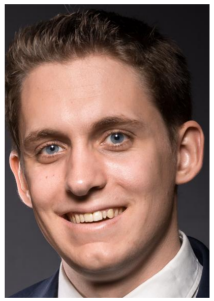
✓means costs attained (synchronization within polylog factors).

[1] B. Lipshitz, MS thesis 2013
[2] T. Wicky, E.S., T. Hoefler, IPDPS 2017
[3] A. Tiskin, FGCS 2007
[4] E.S., J. Demmel, EuroPar 2011
[5] E.S., G. Ballard, T. Hoefler, J. Demmel, SPAA 2017

New algorithms can circumvent lower bounds

For TRSM, we can achieve a lower synchronization/communication cost by performing triangular inversion on diagonal blocks





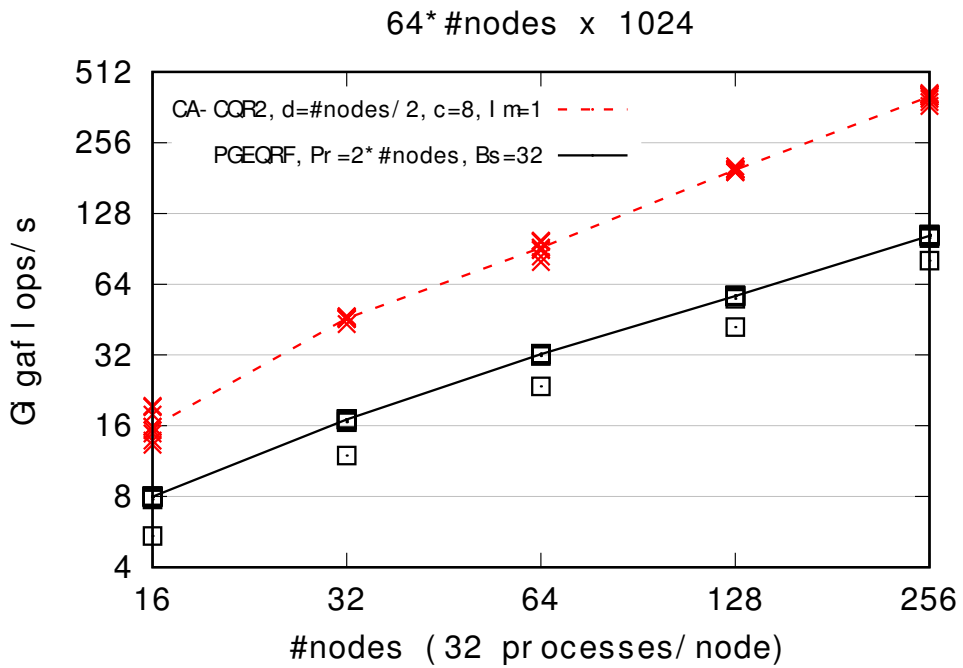
- decreases synchronization cost by $O(p^{2/3})$ on p processors with respect to known algorithms
- optimal communication for any number of right-hand sides
- MS thesis work by Tobias Wicky[1]

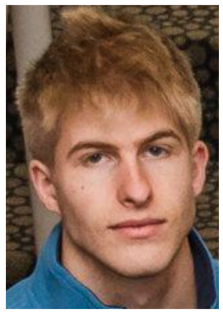
[1] T. Wicky, E.S., T. Hoefler, IPDPS 2017

Cholesky-QR2 for rectangular matrices

Cholesky-QR2[1] with 3D Cholesky gives a practical 3D QR algorithm

- Compute $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R}$ using Cholesky $\boldsymbol{A}^T\boldsymbol{A} = \boldsymbol{R}^T\boldsymbol{R}$
- Correct computed factorization by Cholesky-QR of $\boldsymbol{Q}$
- Attains full accuracy so long as cond$(\boldsymbol{A}) < 1/\sqrt{\epsilon_{\text{mach}}}$





work by Edward Hutter (PhD student at UIUC)

[1] Fukaya T, Nakatsukasa Y, Yanagisawa Y, Yamamoto Y. 2014

Tridiagonalization

Reducing the symmetric matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ to a tridiagonal matrix

$$\boldsymbol{T} = \boldsymbol{Q}^T \boldsymbol{A} \boldsymbol{Q}$$

via a two-sided orthogonal transformation is most costly in diagonalization (eigenvalue computation, SVD similar)

- can be done by successive column QR factorizations

$$\boldsymbol{T} = \underbrace{\boldsymbol{Q_1}^T \cdots \boldsymbol{Q_n}^T}_{\boldsymbol{Q}^T} \boldsymbol{A} \underbrace{\boldsymbol{Q_1} \cdots \boldsymbol{Q_n}}_{\boldsymbol{Q}}$$

- two-sided updates harder to manage than one-sided
- can use n/b QRs on panels of b columns to go to band-width $b+1$
- $b = 1$ gives direct tridiagonalization

Multi-stage tridiagonalization

Writing the orthogonal transformation in Householder form, we get

$$\underbrace{(\boldsymbol{I} - \boldsymbol{U}\boldsymbol{T}\boldsymbol{U}^T)^T}_{\boldsymbol{Q}^T} \boldsymbol{A} \underbrace{(\boldsymbol{I} - \boldsymbol{U}\boldsymbol{T}\boldsymbol{U}^T)}_{\boldsymbol{Q}} = \boldsymbol{A} - \boldsymbol{U}\boldsymbol{V}^T - \boldsymbol{V}\boldsymbol{U}^T$$

where columns of $\boldsymbol{U}$ are Householder vectors and $\boldsymbol{V}$ is

$$\boldsymbol{V}^T = \boldsymbol{T}\boldsymbol{U}^T + \frac{1}{2}\boldsymbol{T}^T\boldsymbol{U}^T \underbrace{\boldsymbol{A}\boldsymbol{U}}_{\text{challenge}} \boldsymbol{T}\boldsymbol{U}^T$$

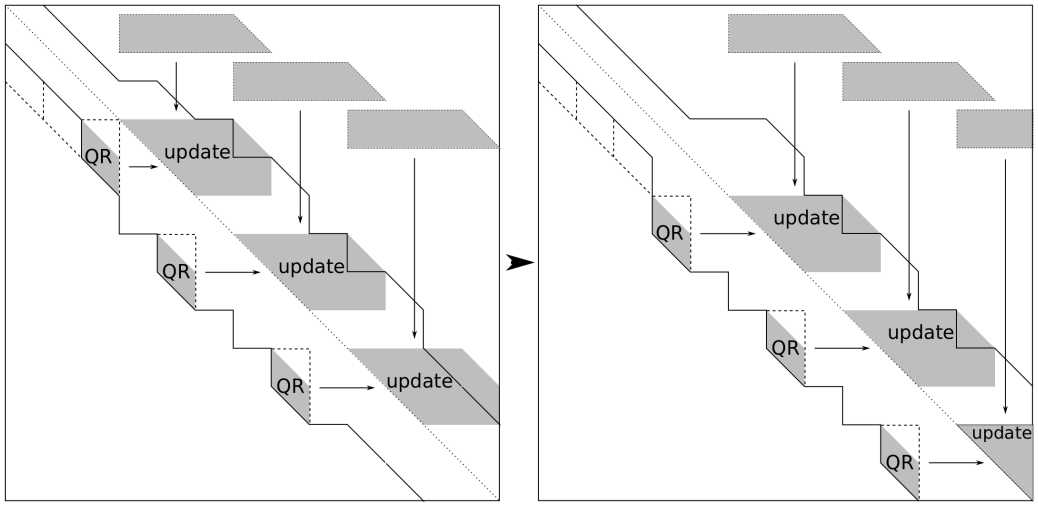
- if $b = 1$, $\boldsymbol{U}$ is a column-vector, and $\boldsymbol{A}\boldsymbol{U}$ is dominated by vertical communication cost (moving $\boldsymbol{A}$ between memory and cache)
- idea: reduce to banded matrix ($b \gg 1$) first[1]

[1] Auckenthaler, Bungartz, Huckle, Krämer, Lang, Willems 2011

Successive band reduction (SBR)

After reducing to a banded matrix, we need to transform the banded matrix to a tridiagonal one

- fewer nonzeros lead to lower computational cost, $F = O(n^2 b/p)$
- however, transformations introduce fill/bulges
- bulges must be chased down the band[1]



- communication- and synchronization-efficient 1D SBR algorithm known for small band-width[2]

[1] Lang 1993; Bischof, Lang, Sun 2000

[2] Ballard, Demmel, Knight 2012

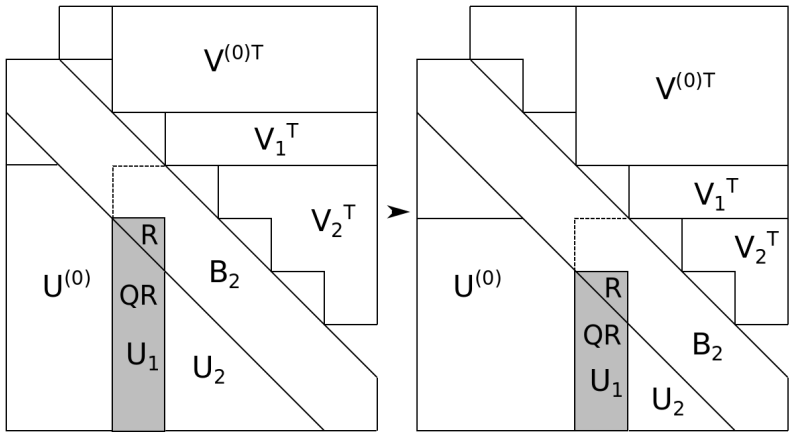
Communication-efficient eigenvalue computation

Previous work (start-of-the-art): two-stage tridiagonalization

- implemented in ELPA, can outperform ScaLAPACK[1]
- with $n = n/\sqrt{p}$, 1D SBR gives $W = O(n^2/\sqrt{p})$, $S = O(\sqrt{p}\log^2(p))$[2]

New results[3]: many-stage tridiagonalization

- $\Theta(\log(p))$ intermediate band-widths to achieve $W = O(n^2/p^{2/3})$
- communication-efficient rectangular QR with processor groups



- 3D SBR (each QR and matrix multiplication update parallelized)

[1] Auckenthaler, Bungartz, Huckle, Krämer, Lang, Willems 2011

[2] Ballard, Demmel, Knight 2012

[3] S., Ballard, Demmel, Hoefler 2017

Symmetric eigensolver results summary

Algorithm	W	Q	S
ScaLAPACK	$n^2/\sqrt{p}$	n^3/p	$n\log(p)$
ELPA	$n^2/\sqrt{p}$	-	$n\log(p)$
two-stage + 1D-SBR	$n^2/\sqrt{p}$	$n^2\log(n)/\sqrt{p}$	$\sqrt{p}(\log^2(p)+\log(n))$
many-stage	$n^2/p^{2/3}$	$n^2\log(p)/p^{2/3}$	$p^{2/3}\log^2 p$

- costs are asymptotic (same computational cost F for eigenvalues)
- W – horizontal (interprocessor) communication
- Q – vertical (memory–cache) communication excluding $W + F/\sqrt{H}$ where H is cache size
- S – synchronization cost (number of supersteps)

Future directions and acknowledgements

Future/ongoing directions in Cyclops development

- General abstractions for tensor decompositions
- Concurrent scheduling of multiple contractions
- Fourier transforms along tensor modes
- Faster/specialized/optimized tensor slicing

Open problems in communication-avoiding matrix factorizations

- Algorithm for SVD with 3D cost without log factor overhead flops
- 3D implementation of SVD and QR with column pivoting

Acknowledgements

- Devin Matthews (UT Austin), Jeff Hammond (Intel Corp.), Grey Ballard, James Demmel (UC Berkeley), Tobias Wicky, Torsten Hoefler (ETH Zurich), Edward Hutter, Zecheng Zhang, Eric Song, Eduardo Yap, Linjian Ma (UIUC)
- Computational resources at NERSC, CSCS, ALCF, NCSA, TACC

Backup slides

Fast algorithms for symmetric tensor contractions

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

- order d (i.e. d modes / indices)
- dimensions n-by-$\cdots$-by-n
- elements $\boldsymbol{T}_{\boldsymbol{i}_1 \ldots \boldsymbol{i}_d} = \boldsymbol{T}_{\boldsymbol{i}}$ where $\boldsymbol{i} \in \{1, \ldots, n\}^d$

We say a tensor is symmetric if for any $j, k \in \{1, \ldots, n\}$

$$\boldsymbol{T}_{\boldsymbol{i}_1 \ldots \boldsymbol{i}_j \ldots \boldsymbol{i}_k \ldots \boldsymbol{i}_d} = \boldsymbol{T}_{\boldsymbol{i}_1 \ldots \boldsymbol{i}_k \ldots \boldsymbol{i}_j \ldots \boldsymbol{i}_d}$$

A tensor is partially-symmetric if such index interchanges are restricted to be within subsets of $\{1, \ldots, n\}$, e.g.

$$\boldsymbol{T}^{ij}_{kl} = \boldsymbol{T}^{ji}_{kl} = \boldsymbol{T}^{ji}_{lk} = \boldsymbol{T}^{ij}_{lk}$$

For any $s, t, v \in \{0, 1, \ldots\}$, a tensor contraction is

$$\forall \boldsymbol{i} \in \{1, \ldots, n\}^s, \boldsymbol{j} \in \{1, \ldots, n\}^t, \quad \boldsymbol{C}_{\boldsymbol{ij}} = \sum_{\boldsymbol{k} \in \{1, \ldots, n\}^v} \boldsymbol{A}_{\boldsymbol{ik}} \boldsymbol{B}_{\boldsymbol{kj}}$$

Symmetric matrix times vector

Lets consider the simplest tensor contraction with symmetry

- let $\boldsymbol{A}$ be an n-by-n symmetric matrix $(\boldsymbol{A}_{ij} = \boldsymbol{A}_{ji})$
- the symmetry is not preserved in matrix-vector multiplication

$$\boldsymbol{c} = \boldsymbol{A} \cdot \boldsymbol{b}$$

$$\boldsymbol{c}_i = \sum_{j=1}^{n} \underbrace{\boldsymbol{A}_{ij} \cdot \boldsymbol{b}_j}_{\text{nonsymmetric}}$$

- generally n^2 additions and n^2 multiplications are performed
- we can perform only $\binom{n+1}{2}$ multiplications using[1]

$$\boldsymbol{c}_i = \sum_{j=1, j\neq i}^{n} \underbrace{\boldsymbol{A}_{ij} \cdot (\boldsymbol{b}_i + \boldsymbol{b}_j)}_{\text{symmetric}} + \underbrace{\left(\boldsymbol{A}_{ii} - \sum_{j=1, j\neq i}^{n} \boldsymbol{A}_{ij}\right) \cdot \boldsymbol{b}_i}_{\text{low-order}}$$

[1] E.S., J. Demmel, 2015

Symmetrized outer product

Consider a rank-2 outer product of vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ of length n into symmetric matrix $\boldsymbol{C}$

$$\boldsymbol{C} = \boldsymbol{a} \cdot \boldsymbol{b}^{\mathsf{T}} + \boldsymbol{b} \cdot \boldsymbol{a}^{\mathsf{T}}$$

$$\boldsymbol{C}_{ij} = \underbrace{\boldsymbol{a}_i \cdot \boldsymbol{b}_j}_{\text{nonsymmetric}} + \underbrace{\boldsymbol{a}_j \cdot \boldsymbol{b}_i}_{\text{permutation}}$$

usually computed via the n^2 multiplications and n^2 additions
new algorithm requires $\binom{n+1}{2}$ multiplications

$$\boldsymbol{C}_{ij} = \underbrace{\underbrace{(\boldsymbol{a}_i + \boldsymbol{a}_j) \cdot (\boldsymbol{b}_i + \boldsymbol{b}_j)}_{\boldsymbol{z}_{ij}}}_{\text{symmetric}} - \underbrace{\underbrace{\boldsymbol{a}_i \cdot \boldsymbol{b}_i}_{\boldsymbol{w}_i} - \underbrace{\boldsymbol{a}_j \cdot \boldsymbol{b}_j}_{\boldsymbol{w}_j}}_{\text{low-order}}$$

Symmetrized matrix multiplication

For symmetric matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, compute

$$\boldsymbol{C}_{ij} = \sum_{k=1}^{n} \Big(\underbrace{\boldsymbol{A}_{ik} \cdot \boldsymbol{B}_{kj}}_{\text{nonsymmetric}} + \underbrace{\boldsymbol{A}_{jk} \cdot \boldsymbol{B}_{ki}}_{\text{permutation}} \Big)$$

New algorithm requires $\binom{n+2}{3} + \binom{n}{2}$ multiplications rather than n^3, based on

$$\boldsymbol{C}_{ij} = \sum_{k=1}^{n} \Big(\underbrace{\boldsymbol{A}_{ik} \cdot \boldsymbol{B}_{kj}}_{\text{nonsymmetric}} + \underbrace{\boldsymbol{A}_{jk} \cdot \boldsymbol{B}_{ki}}_{\text{transpose}} \Big)$$

$$= \sum_{k=1}^{n} \underbrace{(\boldsymbol{A}_{ij} + \boldsymbol{A}_{ik} + \boldsymbol{A}_{jk}) \cdot (\boldsymbol{B}_{ij} + \boldsymbol{B}_{kj} + \boldsymbol{B}_{ki})}_{\hat{\boldsymbol{Z}}_{ijk}} - \underbrace{\sum_{k=1}^{n} \boldsymbol{A}_{ik} \cdot \boldsymbol{B}_{ik}}_{\boldsymbol{w}_i} - \underbrace{\sum_{k=1}^{n} \boldsymbol{A}_{jk} \cdot \boldsymbol{B}_{jk}}_{\boldsymbol{w}_j}$$

$$- n\boldsymbol{A}_{ij} \cdot \boldsymbol{B}_{ij} - \boldsymbol{A}_{ij} \cdot \Big(\underbrace{\sum_{k=1}^{n} \boldsymbol{B}_{ki}}_{\boldsymbol{B}_i^{(1)}} + \underbrace{\sum_{k=1}^{n} \boldsymbol{B}_{kj}}_{\boldsymbol{B}_j^{(1)}} \Big) - \Big(\underbrace{\sum_{k=1}^{n} \boldsymbol{A}_{ki}}_{\boldsymbol{A}_i^{(1)}} + \underbrace{\sum_{k=1}^{n} \boldsymbol{A}_{kj}}_{\boldsymbol{A}_j^{(1)}} \Big) \cdot \boldsymbol{B}_{ij}.$$

Symmetrized tensor contraction

Generally consider any symmetric tensor contraction for $s, t, v \in \{0, 1, \ldots\}$

$$\forall \boldsymbol{i} \in \{1, \ldots, n\}^s, \boldsymbol{j} \in \{1, \ldots, n\}^t, \boldsymbol{C_{ij}} = \sum_{\boldsymbol{k} \in \{1,\ldots,n\}^v} \boldsymbol{A_{ik}} \boldsymbol{B_{kj}} + \text{permutations}$$

- best previous algorithms used roughly $\binom{n}{s}\binom{n}{t}\binom{n}{v}$ multiplications, new algorithm requires roughly $\binom{n}{s+t+v}$ multiplications
- communication lower bounds for fast symmetric contractions
  - surprising negative result – when $s + t + v \geq 4$ and $s \neq t \neq v$ asymptotically more communication necessary for new algorithm!
- algorithm can be nested in the case of partially-symmetric contractions, leads to a reduction in cost – manyfold cost improvements in some high-order quantum chemistry methods