Scalable algebraic operations for tensors and graphs

Edgar Solomonik

University of Illinois at Urbana-Champaign

July 31, 2017

Tensor abstractions for parallel computing

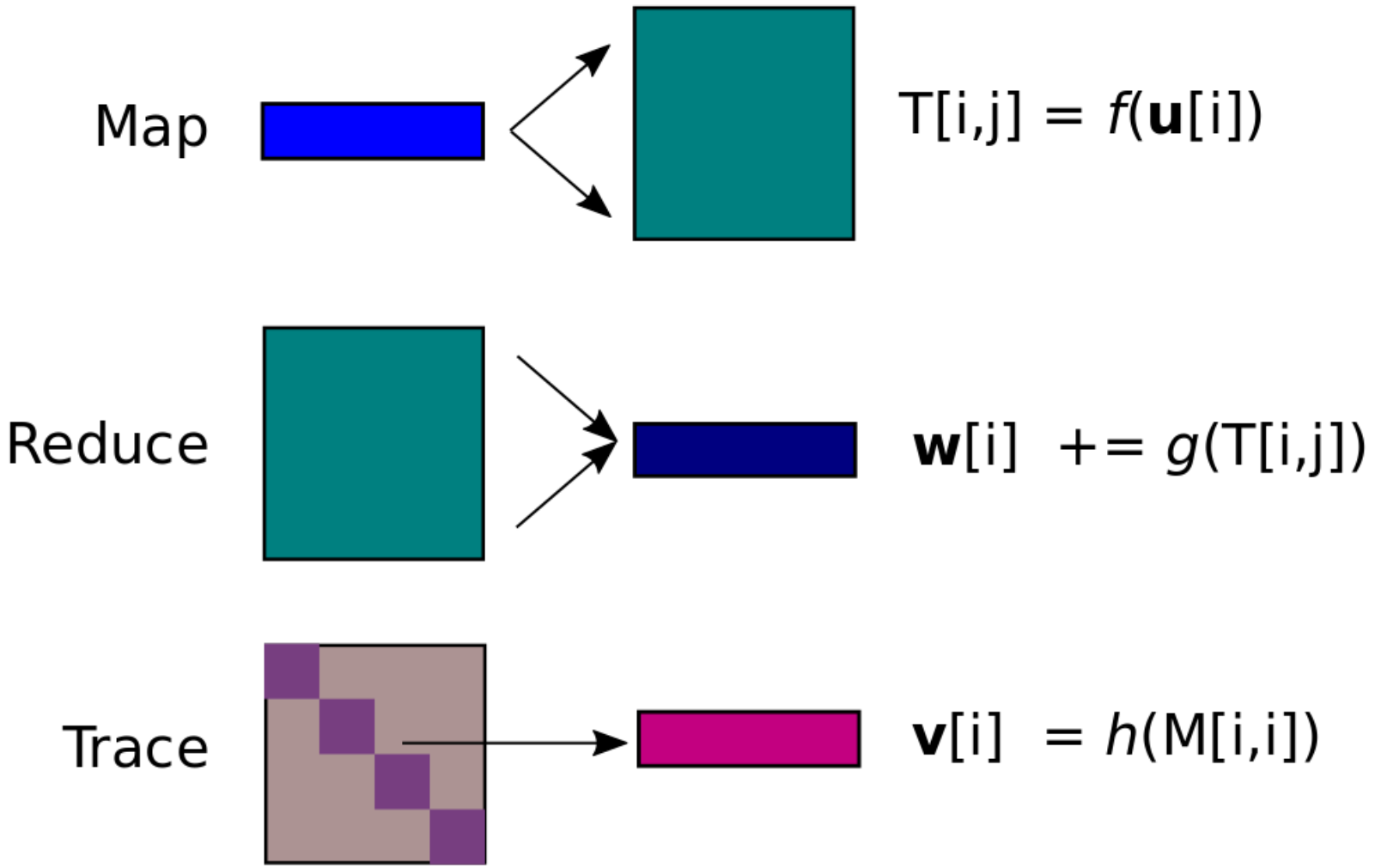
Algebraic tensor operations are a natural language for massive datasets

- 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)

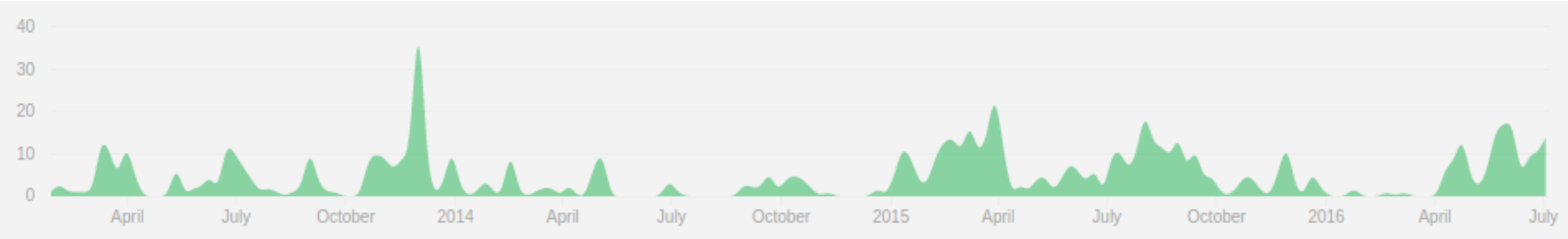
- distributed-memory symmetric/sparse tensors as C++ objects

```
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.

A library for tensor computations

Cyclops Tensor Framework

- contraction/summation/functions of tensors
- distributed symmetric-packed/sparse storage via cyclic layout
- parallelization via MPI+OpenMP(+CUDA)

Jacobi iteration (solves $Ax = b$ iteratively) example code snippet

```
Vector<> Jacobi(Matrix<> A, Vector<> b, int n){
  ... // split A = R + diag(1./d)
  do {
    x["i"] = d["i"]*(b["i"]-R["ij"]*x["j"]);
    r["i"] = b["i"]-A["ij"]*x["j"]; // compute residual
  } while (r.norm2() > 1.E-6); // check for convergence
  return x;
}
```

A library for tensor computations

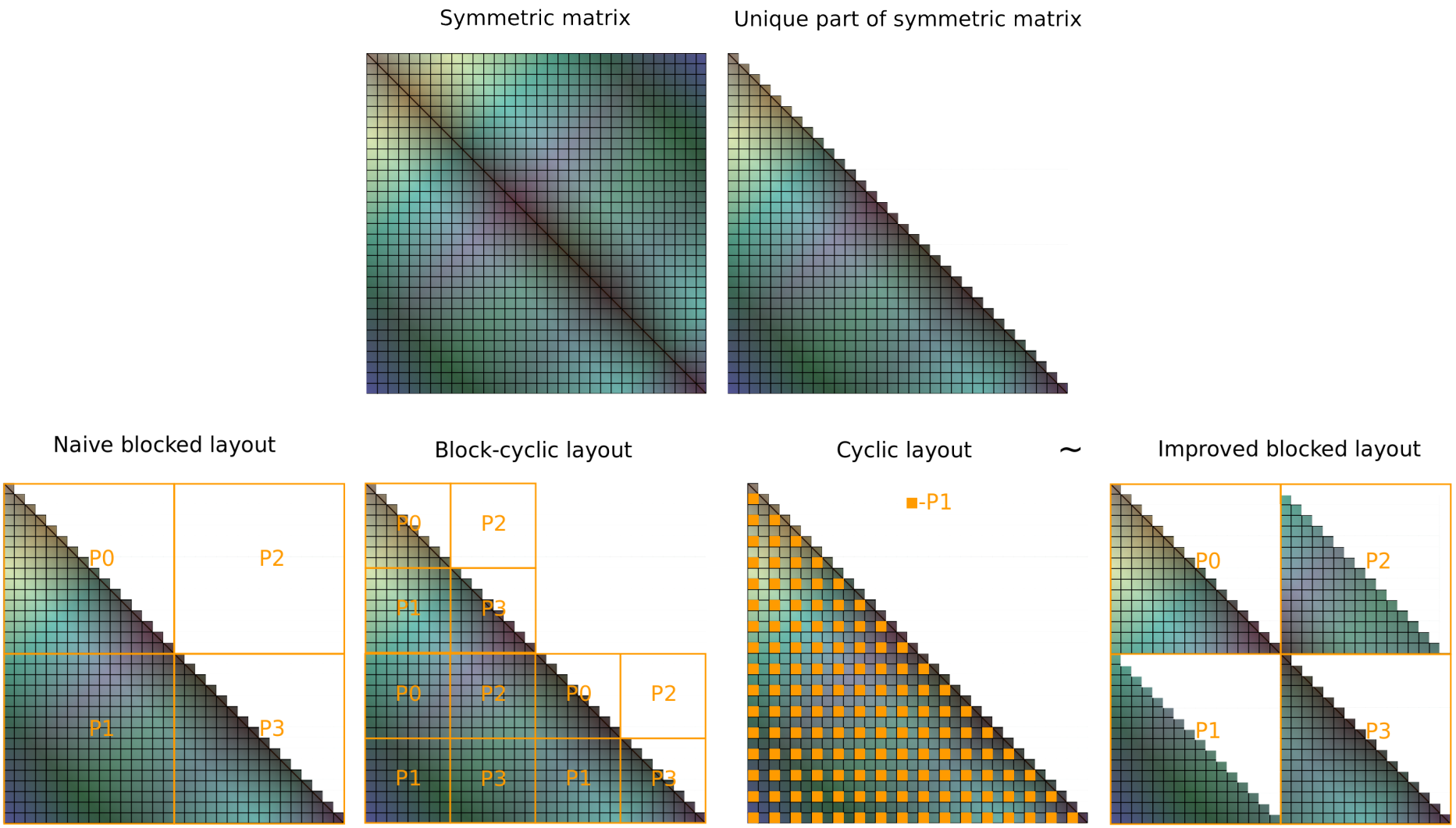
Cyclops Tensor Framework

- contraction/summation/functions of tensors
- distributed symmetric-packed/sparse storage via cyclic layout
- parallelization via MPI+OpenMP(+CUDA)

Jacobi iteration (solves $Ax = b$ iteratively) example code snippet

```
Vector<> Jacobi(Matrix<> A, Vector<> b, int n){
  Matrix<> R(A);
  R["ii"] = 0.0;
  Vector<> x(n), d(n), r(n);
  Function<> inv([](double & d){ return 1./d; });
  d["i"] = inv(A["ii"]); // set d to inverse of diagonal of A
  do {
    x["i"] = d["i"]*(b["i"]-R["ij"]*x["j"]);
    r["i"] = b["i"]-A["ij"]*x["j"]; // compute residual
  } while (r.norm2() > 1.E-6); // check for convergence
  return x;
}
```

Balancing load via a cyclic data decomposition

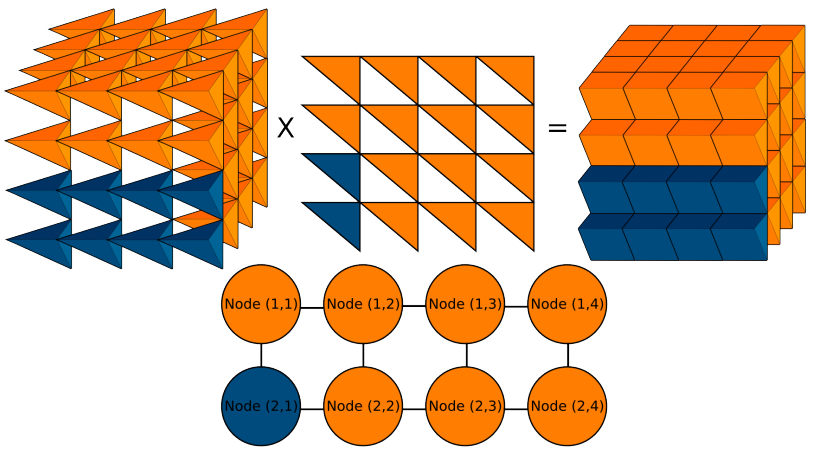


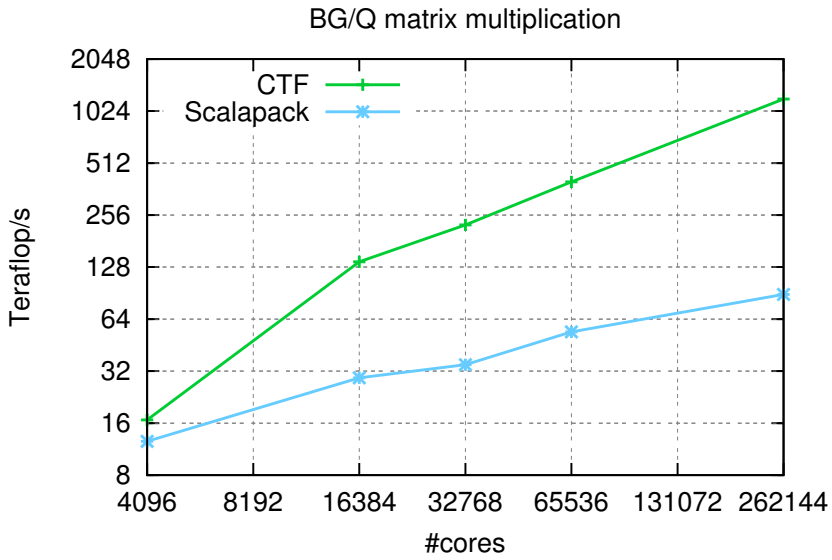
for sparse tensors, a cyclic layout also provides a load-balanced distribution w.h.p. if the number of nonzeros is sufficiently large

CTF parallel scalability

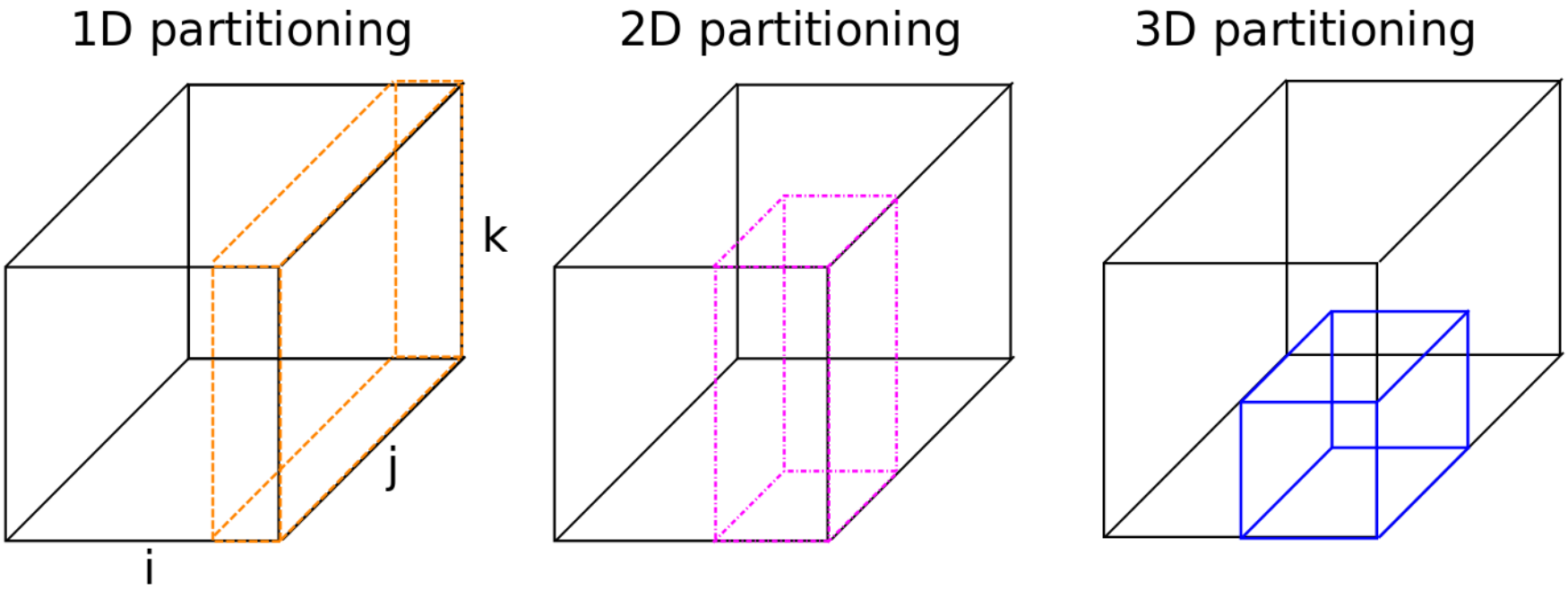
CTF is tuned for massively-parallel architectures

- 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

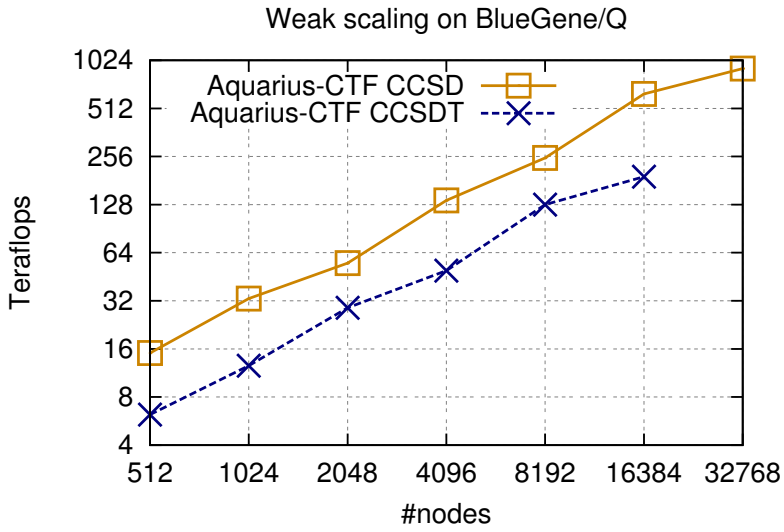
Extracted from Aquarius (lead by Devin Matthews)
https://github.com/devinamatthews/aquarius

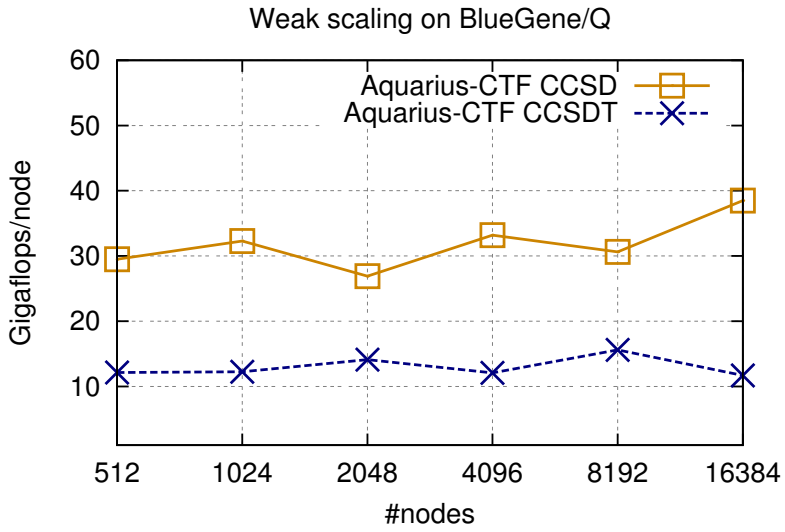
```
FMI["mi"]     += 0.5*WMNEF["mnef"]*T2["efin"];
WMNIJ["mnij"] += 0.5*WMNEF["mnef"]*T2["efij"];
FAE["ae"]     -= 0.5*WMNEF["mnef"]*T2["afmn"];
WAMEI["amei"] -= 0.5*WMNEF["mnef"]*T2["afin"];

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"];
```

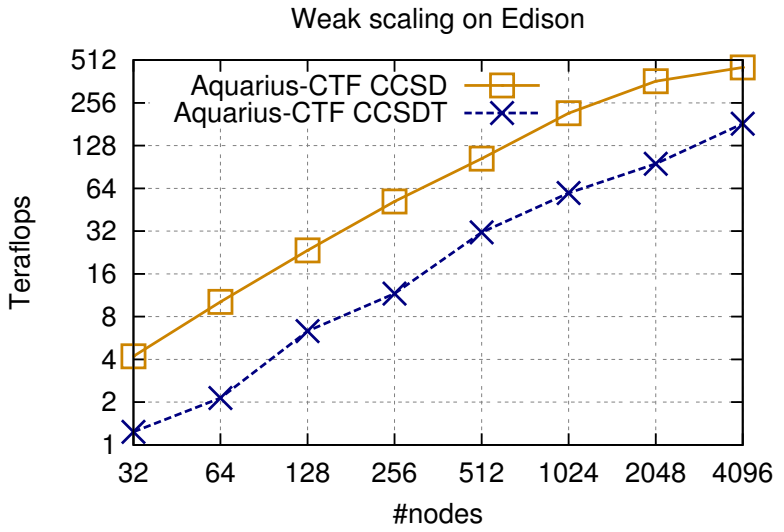
Dense tensor application: coupled cluster performance

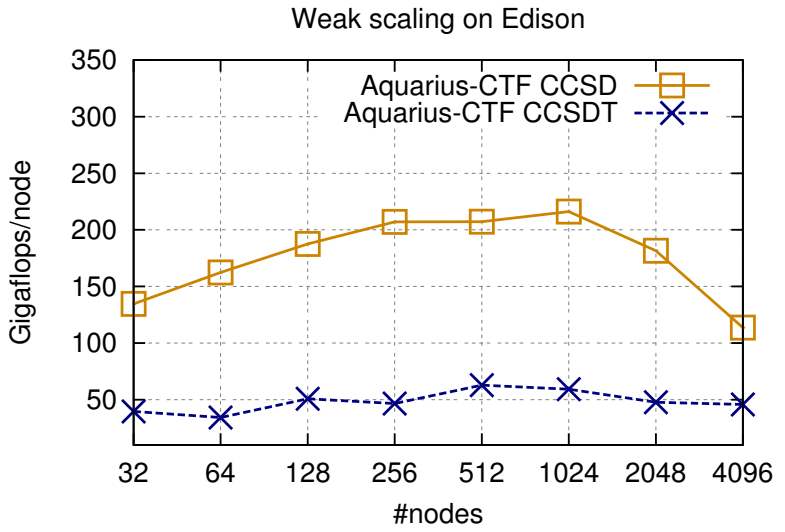
CCSD up to 55 (50) water molecules with cc-pVDZ

CCSDT up to 10 water molecules with cc-pVDZ





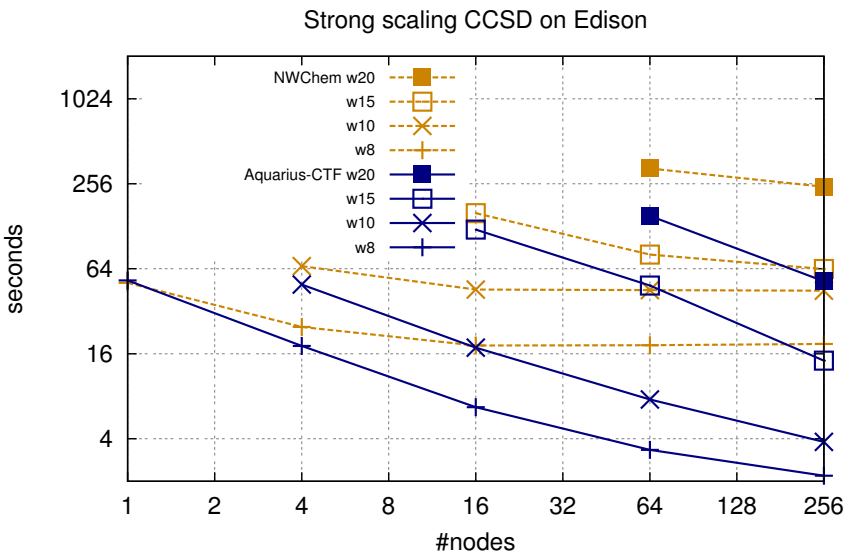


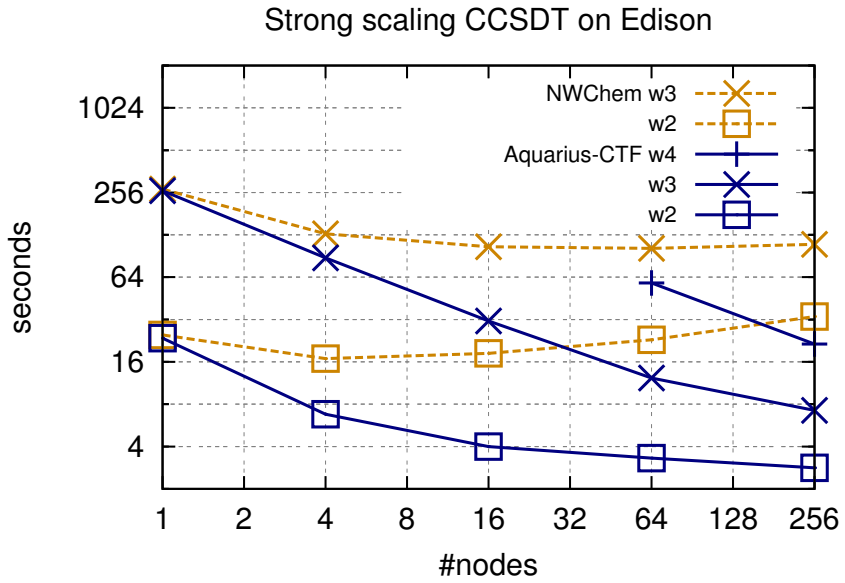


Comparison with NWChem

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



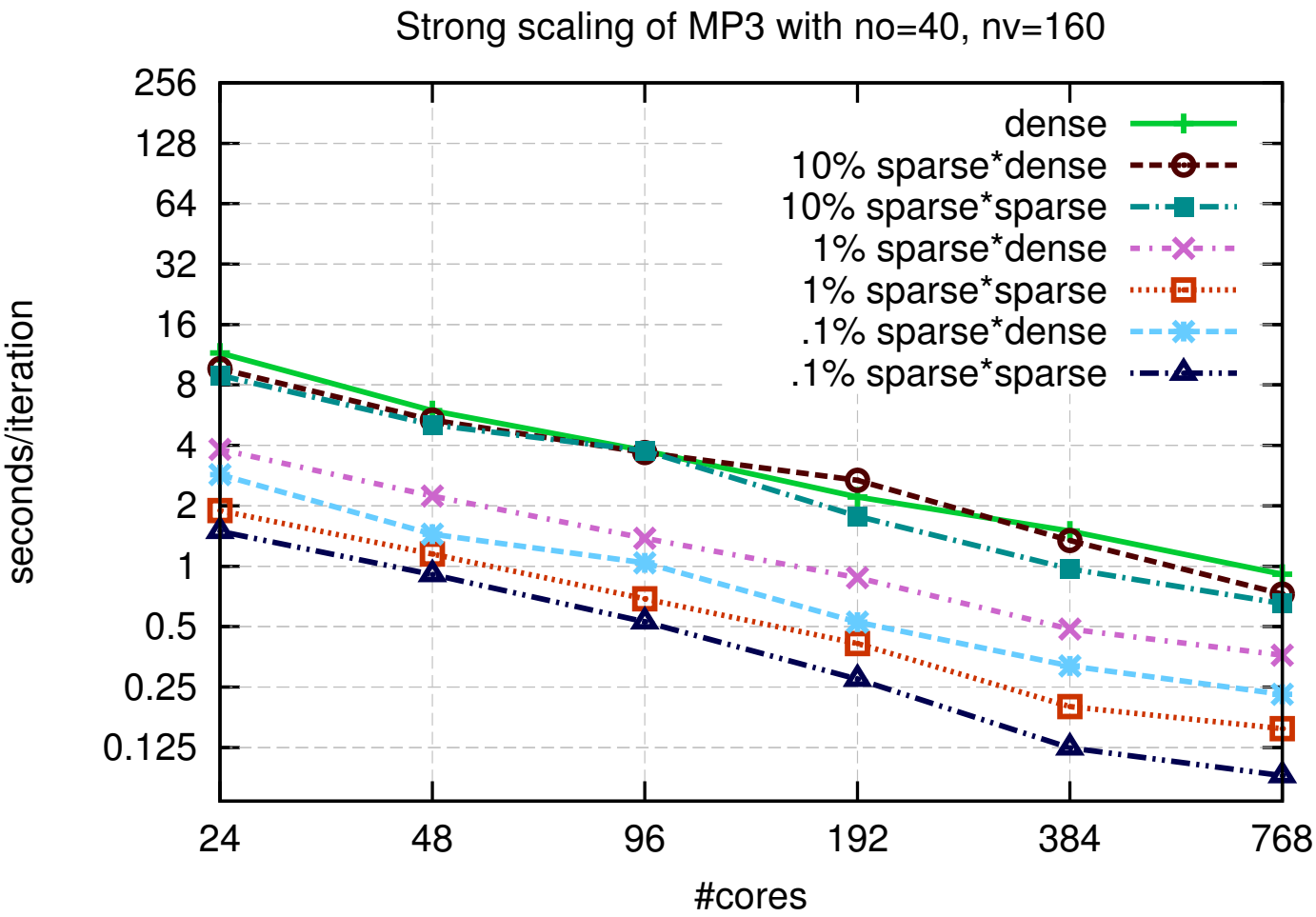


Sparse tensor application: MP3 calculation

```
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"];
```

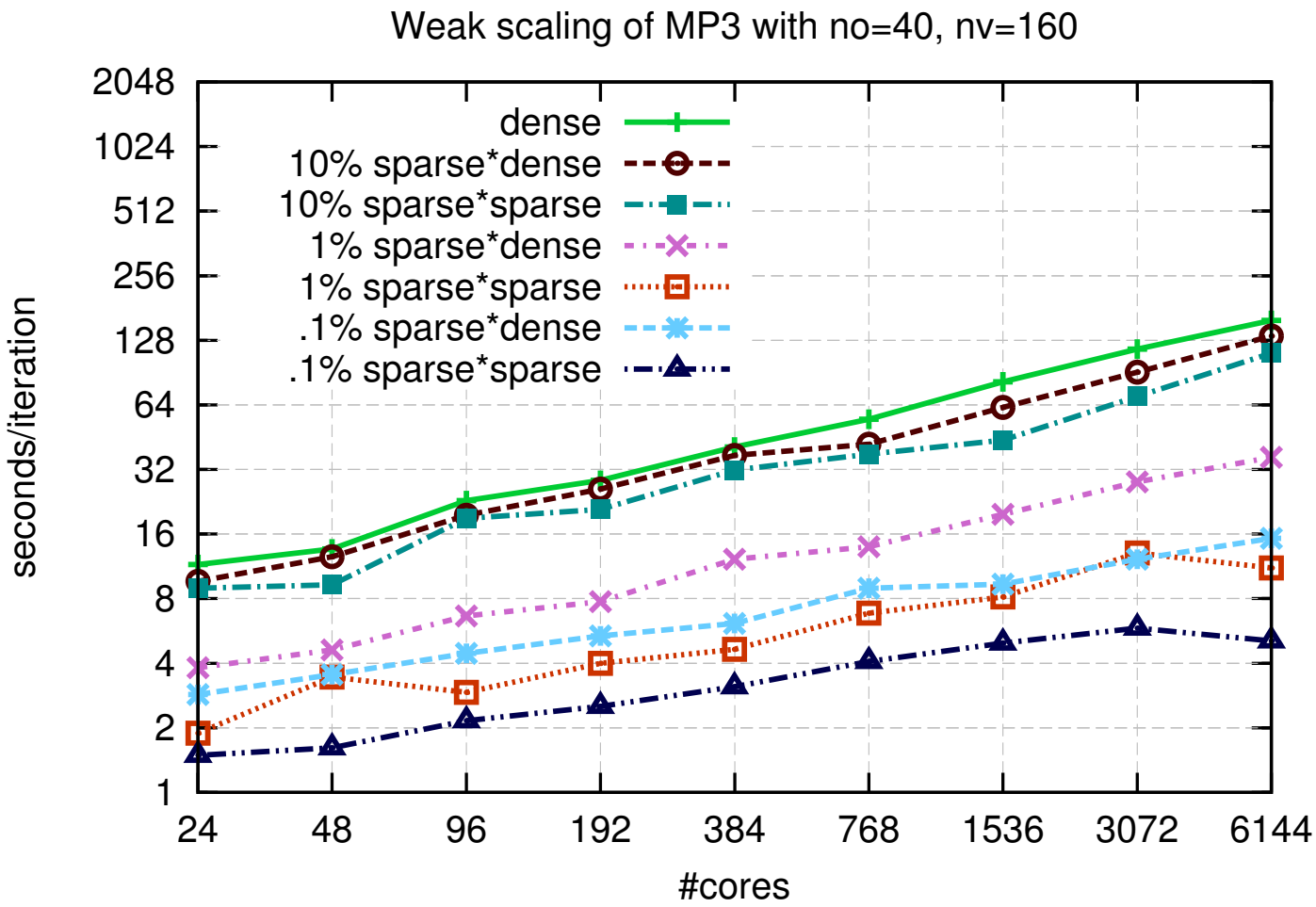
Sparse tensor application: strong scaling

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

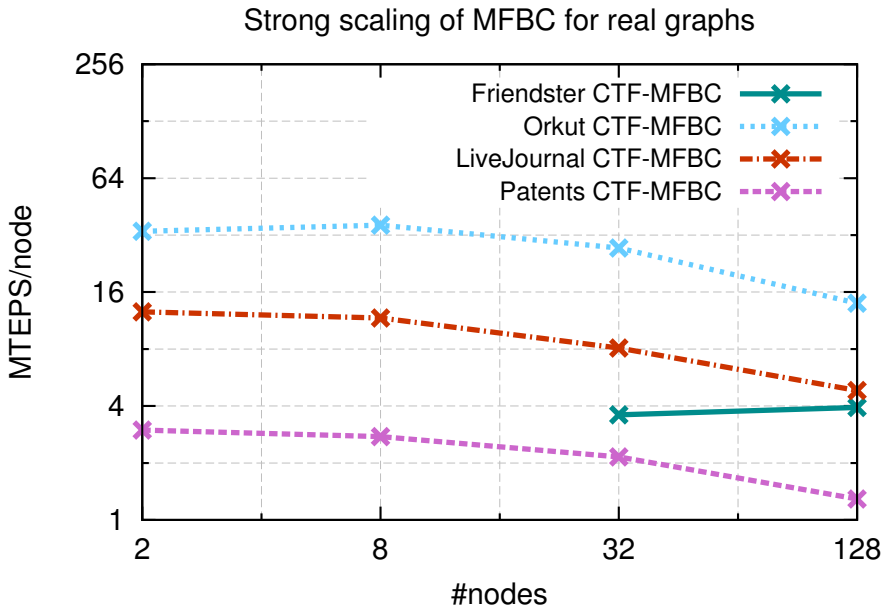
Special operator application: betweenness centrality

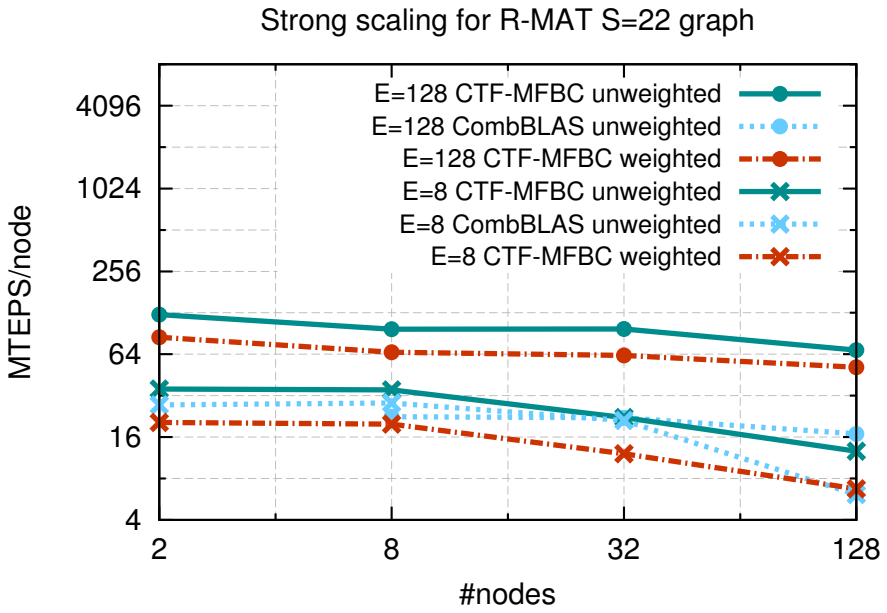
Betweenness centrality code snippet, for k of n nodes

```
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"]);
  ...
}
```

CTF for betweenness centrality

Betweenness centrality using sparse matrix multiplication (SpGEMM) with operations on special monoids





Friendster has 66 million vertices and 1.8 billion edges (results on Blue Waters, Cray XE6)

CTF status and explorations

Much ongoing work and future directions in CTF

- other applications
  - algebraic multigrid: easy implementation but structure-obliviousness costly
  - spectral element methods: unassembled matrix-vector products and gather-scatter via tensor contractions
  - neural networks: tensor structure especially useful for CNNs
- ongoing and future work
  - recent: hook-ups for conversion to/from ScaLAPACK format
  - active: development of Python interface (einsum and slicing work)
  - active: tensor networks and tensor factorization
  - future: performance improvement for batched tensor operations
  - future: predefined output sparsity for contractions
- 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

Communication-synchronization wall

To analyze parallel algorithms, we consider costs along the critical path of the execution schedule[1]

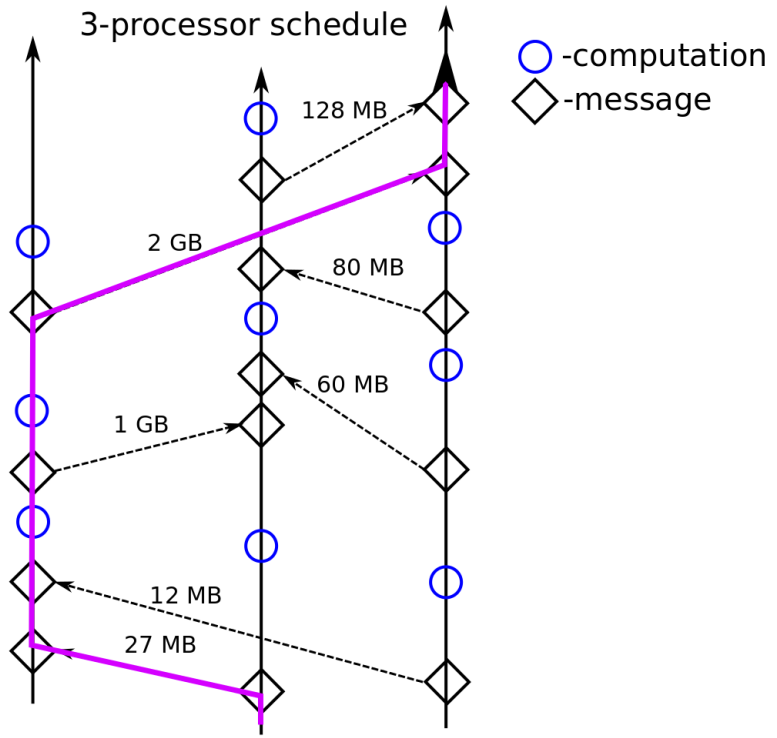
- F – computation cost
- W – horizontal communication cost
- S – synchronization cost

We can show a commonality between

- Cholesky of an $n \times n$ matrix and
- n steps of a 9-pt stencil:

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

regardless of #processors[1]

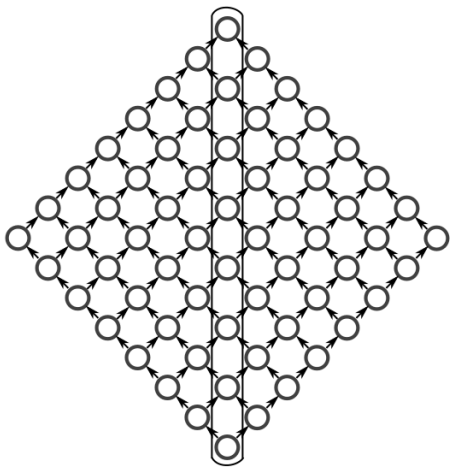


[1] E.S., E. Carson, N. Knight, J. Demmel, TOPC 2016

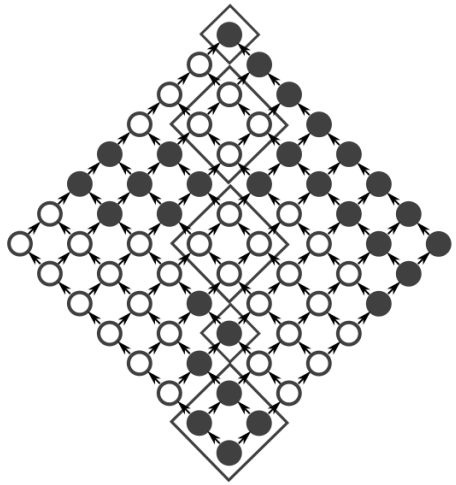
Tradeoffs in the diamond DAG

Computation vs synchronization tradeoff for the $n \times n$ diamond DAG,[2]

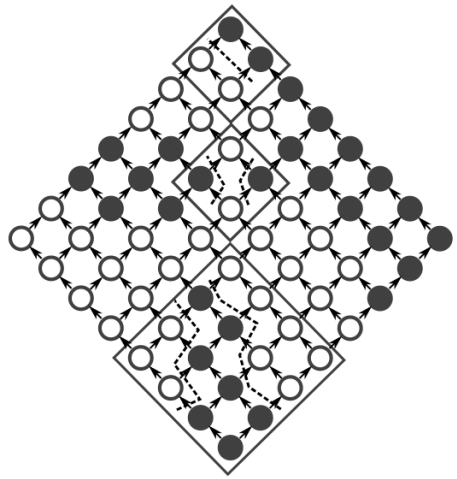
$$F \cdot S = \Omega(n^2)$$



Dependency chain P



Monochrome dependency intervals



Multicolored dependency intervals

In this DAG, vertices denote scalar computations in an algorithm

[2] 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,*

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

Synchronization-communication wall in iterative methods

The theorem can be applied to sparse iterative methods on regular grids. For computing s applications of a $(2m+1)^d$-point stencil,

$$F_{\mathsf{St}} \cdot S_{\mathsf{St}}^d = \Omega\left(m^{2d} \cdot s^{d+1}\right), \qquad W_{\mathsf{St}} \cdot S_{\mathsf{St}}^{d-1} = \Omega\left(m^d \cdot s^d\right)$$

while s-step methods reduce synchronization, for large s they require asymptotically more communication.

The lower bound is attained by s-step methods when s approaches the dimension of each processor's local subgrid.

A more scalable algorithm for TRSM

For Cholesky factorization with p processors, parallel schedules can attain

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

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

triangular solve	square TRSM ✓[3]	rectangular TRSM ✓[4]
LU with pivoting	pairwise pivoting ✓[5]	tournament pivoting ✓[6]
QR factorization	Givens on square ✓[3]	Householder on rect. ✓[7]
SVD	singular values only ✓[5]	singular vectors X

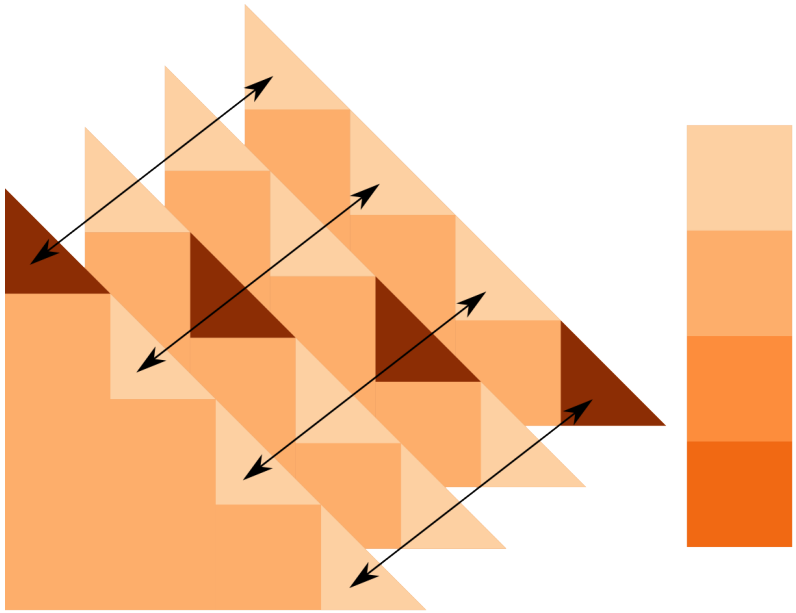
✓ means costs attained (synchronization within polylogarithmic factors).

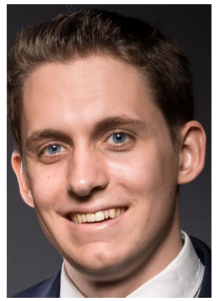
Ongoing work on QR with column pivoting

[3] B. Lipshitz, MS thesis 2013
[4] T. Wicky, E.S., T. Hoefler, IPDPS 2017
[5] A. Tiskin, FGCS 2007
[6] E.S., J. Demmel, EuroPar 2011
[7] 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[8]

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

Improving scalability for iterative methods

Randomized-projection methods have potential to significantly improve scalability over iterative Krylov subspace methods

- key idea: replace sparse mat-vecs with sparse mat-muls
- define $n \times (k+10)$ Gaussian random matrix $\mathbf{X}$
- $\mathbf{AX}$ gives a good representation of the kernel of $\mathbf{A}$
- accuracy can be improved exponentially with q[9]

$$(\mathbf{A}\mathbf{A}^T)^q\mathbf{A}\mathbf{X}$$

- many related results with high potential for efficiency (e.g. randomized column pivoting for QR [10])

[9] N. Halko, P.G. Martinsson, J.A. Tropp, SIAM Review 2011

[10] P.G. Martinsson, G. Quintana Orti, N. Heavner. R. van de Geijn, SIAM 2017

Our CCSD factorization

$$
\begin{aligned}
\tilde{W}_{ei}^{mn} &= v_{ei}^{mn} + \sum_f v_{ef}^{mn} t_i^f, \\
\tilde{W}_{ij}^{mn} &= v_{ij}^{mn} + P_j^i \sum_e v_{ie}^{mn} t_j^e + \frac{1}{2} \sum_{ef} v_{ef}^{mn} \tau_{ij}^{ef}, \\
\tilde{W}_{ie}^{am} &= v_{ie}^{am} - \sum_n \tilde{W}_{ei}^{mn} t_n^a + \sum_f v_{ef}^{ma} t_i^f + \frac{1}{2} \sum_{nf} v_{ef}^{mn} t_{in}^{af}, \\
\tilde{W}_{ij}^{am} &= v_{ij}^{am} + P_j^i \sum_e v_{ie}^{am} t_j^e + \frac{1}{2} \sum_{ef} v_{ef}^{am} \tau_{ij}^{ef}, \\
z_i^a &= f_i^a - \sum_m \tilde{F}_i^m t_m^a + \sum_e f_e^a t_i^e + \sum_{em} v_{ei}^{ma} t_m^e + \sum_{em} v_{im}^{ae} \tilde{F}_e^m + \frac{1}{2} \sum_{efm} v_{ef}^{am} \tau_{im}^{ef} \\
&- \frac{1}{2} \sum_{emn} \tilde{W}_{ei}^{mn} t_{mn}^{ea}, \\
z_{ij}^{ab} &= v_{ij}^{ab} + P_j^i \sum_e v_{ie}^{ab} t_j^e + P_b^a P_j^i \sum_{me} \tilde{W}_{ie}^{am} t_{mj}^{eb} - P_b^a \sum_m \tilde{W}_{ij}^{am} t_m^b \\
&+ P_b^a \sum_e \tilde{F}_e^a t_{ij}^{eb} - P_j^i \sum_m \tilde{F}_i^m t_{mj}^{ab} + \frac{1}{2} \sum_{ef} v_{ef}^{ab} \tau_{ij}^{ef} + \frac{1}{2} \sum_{mn} \tilde{W}_{ij}^{mn} \tau_{mn}^{ab},
\end{aligned}
$$

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

QR factorization of tall-and-skinny matrices

Consider the reduced factorization $\mathbf{A} = \mathbf{QR}$ with $\mathbf{A}, \mathbf{Q} \in \mathbb{R}^{m \times n}$ and $\mathbf{R} \in \mathbb{R}^{n \times n}$ when $m \gg n$ (in particular $m \geq np$)

- $\mathbf{A}$ is tall-and-skinny, each processor owns a block of rows
- Householder-QR requires $S = \Theta(n)$ supersteps, $W = O(n^2)$
- Cholesky-QR2, TSQR, and HR-TSQR require $S = \Theta(\log(p))$ supersteps
  - Cholesky-QR2[11]: stable so long as $\kappa(\mathbf{A}) \leq 1/\sqrt{\epsilon}$, $W = O(n^2)$

$$\mathbf{L} = \text{Chol}(\mathbf{A}^T\mathbf{A}), \mathbf{Z} = \mathbf{A}\mathbf{L}^{-T}, \bar{\mathbf{L}} = \text{Chol}(\mathbf{Z}^T\mathbf{Z}), \mathbf{Q} = \mathbf{Z}\bar{\mathbf{L}}^{-T}, \mathbf{R} = \bar{\mathbf{L}}^T\mathbf{L}^T$$

  - TSQR[12]: row-recursive divide-and-conquer, $W = O(n^2 \log(p))$

$$\begin{bmatrix}\mathbf{Q_1R_1}\\ \mathbf{Q_2R_2}\end{bmatrix} = \begin{bmatrix}\text{TSQR}(\mathbf{A_1})\\ \text{TSQR}(\mathbf{A_2})\end{bmatrix}, [\mathbf{Q_{12}}, \mathbf{R}] = \text{QR}\left(\begin{bmatrix}\mathbf{R_1}\\ \mathbf{R_2}\end{bmatrix}\right), \mathbf{Q} = \begin{bmatrix}\mathbf{Q_1} & \mathbf{0}\\ \mathbf{0} & \mathbf{Q_2}\end{bmatrix}\mathbf{Q_{12}}$$

  - TSQR-HR[13]: TSQR with Householder-reconstruction, $W = O(n^2 \log(p))$

[11] Yamamoto, Nakatsukasa, Yanagisawa, Fukaya 2015

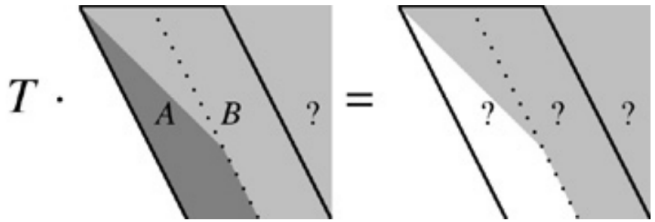
[12] Demmel, Grigori, Hoemmen, Langou 2012

[13] Ballard, Demmel, Grigori, Jacquelin, Nguyen, S. 2014

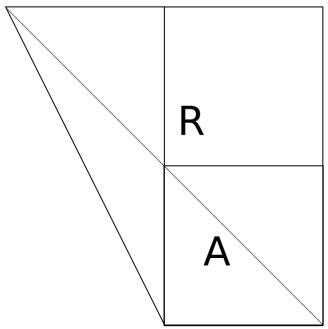
QR factorization of square matrices

Square matrix QR algorithms generally use 1D QR for panel factorization

- algorithms in ScaLAPACK, Elemental, DPLASMA use 2D layout, generally achieve $W = O(n^2/\sqrt{p})$ cost
- Tiskin's 3D QR algorithm[14] achieves $W = O(n^2/p^{2/3})$ communication



- however, requires slanted-panel matrix embedding



which is highly inefficient for rectangular (tall-and-skinny) matrices

[14] Tiskin 2007, "Communication-efficient generic pairwise elimination"

Communication-avoiding rectangular QR

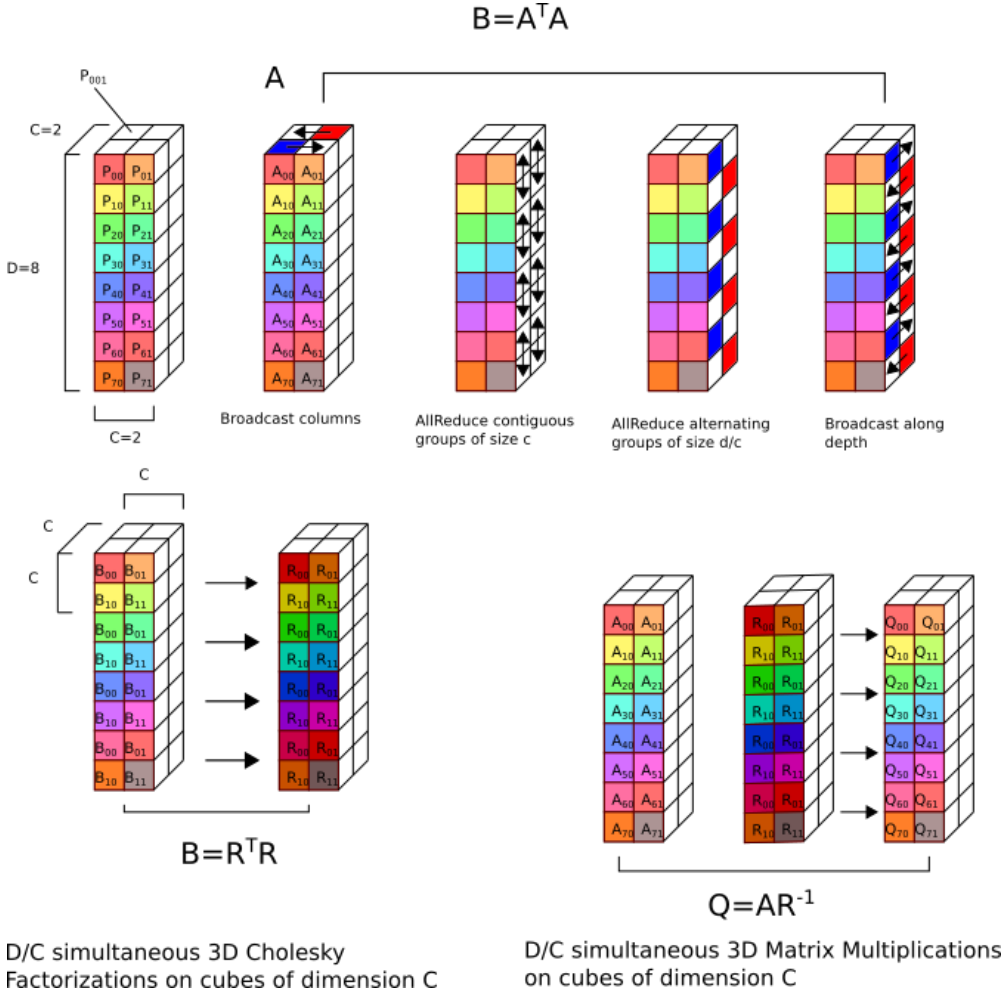
For $\mathbf{A} \in \mathbb{R}^{m \times n}$ existing algorithms are optimal when $m = n$ and $m \gg n$

- cases with $n < m < np$ underdetermined equations are important
- new algorithm
  - subdivide p processors into m/n groups of pn/m processors
  - perform row-recursive QR (TSQR) with tree of height $\log_2(m/n)$
  - compute each tree-node elimination QR$\left(\begin{bmatrix} \mathbf{R_1} \\ \mathbf{R_2} \end{bmatrix} \right)$ using Tiskin's QR with pn/m or more processors
- note: interleaving rows of $\mathbf{R_1}$ and $\mathbf{R_2}$ gives a slanted panel!
- obtains ideal communication cost for any m, n, generally

$$W = O\left(\left(\frac{mn^2}{p} \right)^{2/3} \right)$$

Cholesky-QR2 for rectangular matrices

Cholesky-QR2 with 3D Cholesky provides a simple 3D QR algorithm for well-conditioned rectangular matrices



work by Edward Hutter (PhD student at UIUC)

Tridiagonalization

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

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

via a two-sided orthogonal transformation is most costly in diagonalization

- can be done by successive column QR factorizations

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

- 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

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

where $\mathbf{U}$ are Householder vectors and $\mathbf{V}$ is

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

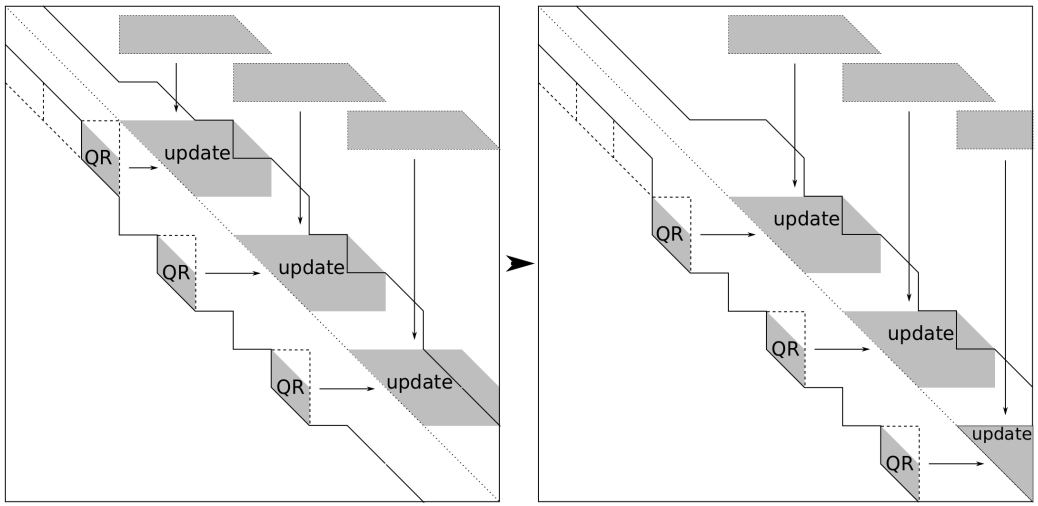
- when performing two-sided updates, computing $\mathbf{AU}$ dominates cost
- if $b = 1$, $\mathbf{U}$ is a column-vector, and $\mathbf{AU}$ is dominated by vertical communication cost (moving $\mathbf{A}$ between memory and cache)
- idea: reduce to banded matrix ($b \gg 1$) first[15]

[15] 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[16]



- communication- and synchronization-efficient 1D SBR algorithm known for small band-width[17]

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

[17] Ballard, Demmel, Knight 2012

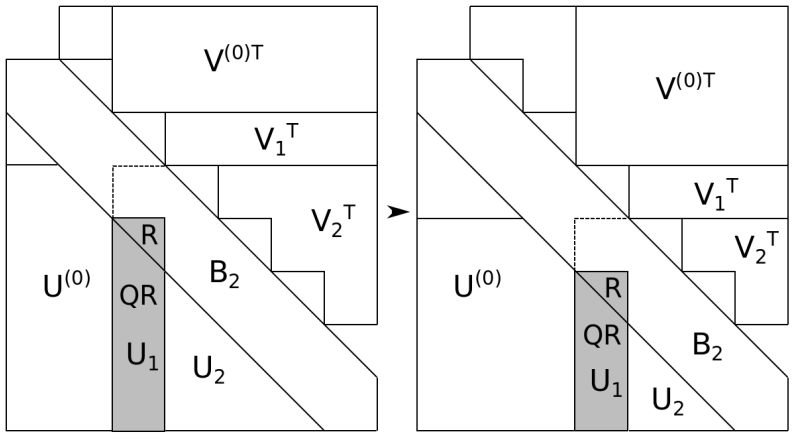
Communication-efficient eigenvalue computation

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

- implemented in ELPA, can outperform ScaLAPACK[18]
- with $n = n/\sqrt{p}$, 1D SBR gives $W = O(n^2/\sqrt{p})$, $S = O(\sqrt{p}\log^2(p))$[19]

New results[20]: many-stage tridiagonalization

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



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

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

[19] Ballard, Demmel, Knight 2012

[20] 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 $W + F/\sqrt{H}$
- S – synchronization cost (number of supersteps)