A parallel tensor framework for Coupled Cluster

Edgar Solomonik[1]

James Demmel[1], Jeff Hammond[2], Devin Matthews[3]

1. UC Berkeley 2. Argonne National Laboratory 3. UT Austin

July, 2012

Outline

Communication costs more than computation

Communication happens off-chip and on-chip and incurs two costs

- latency - time per message
- bandwidth - amount of data per unit time

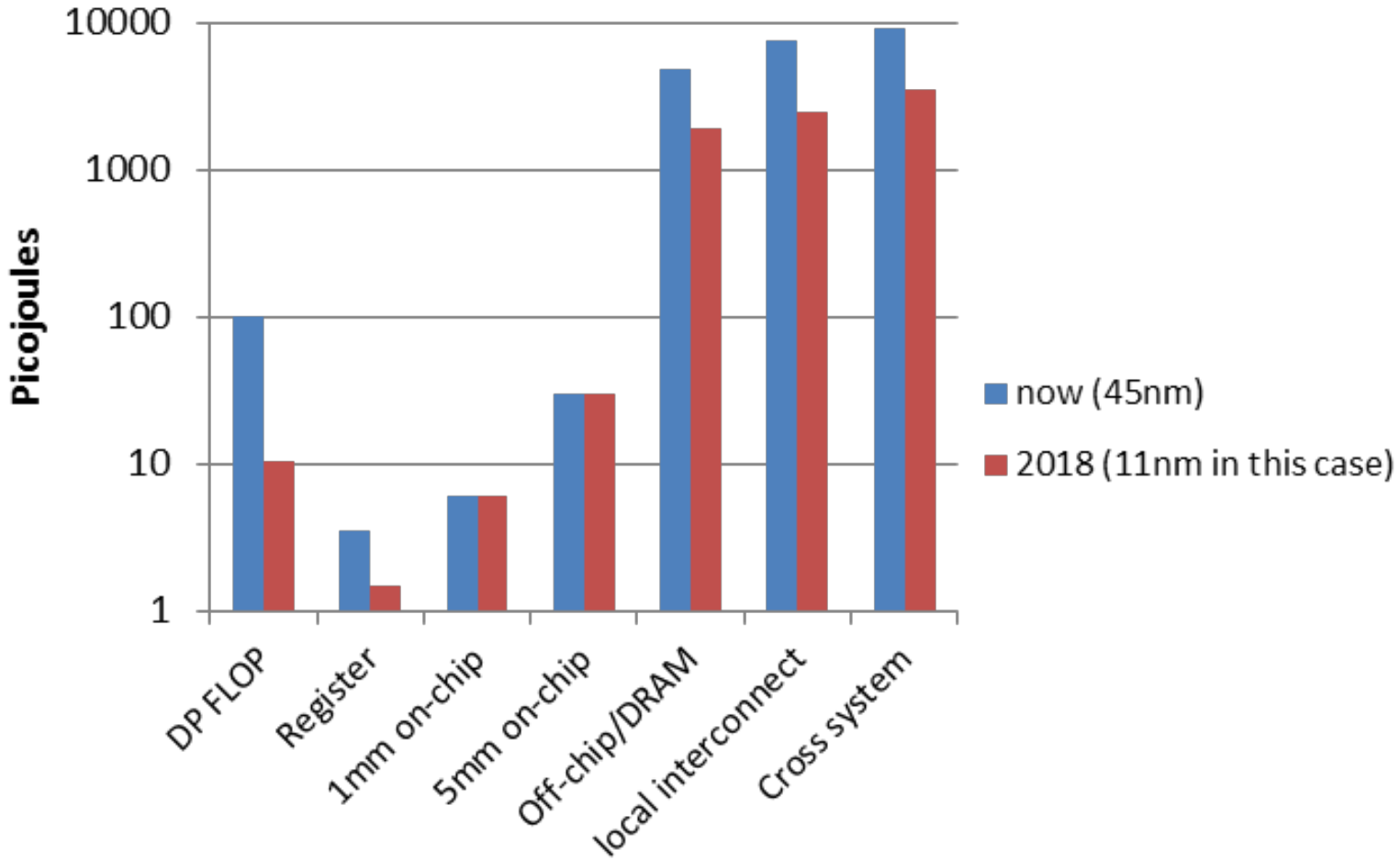
These costs are becoming more expensive relative to flops

Table: Annual improvements

time per flop		bandwidth	latency
59%	network	26%	15%
	DRAM	23%	5%

Source: James Demmel [FOSC]

Communication takes more energy than computation



Source: John Shalf (LBNL)

Sequential communication lower bounds

We study asymptotic communication cost of matrix multiplication

$$c_{ij} = \sum_k a_{ik} \cdot b_{kj}$$

by considering the 3D cuboid computational graph G. The data dependencies of any subset $R \subset G$ are given by projections onto the three faces which represent **A**, **B**, and **C**.

A theorem by Loomis and Whitney: if $|R| = \Theta(M^{3/2})$, the size of some projection is $\Omega(M)$. Think of M as the 'local memory size' which fits computation chunk R. This gives a lower bound on the bandwidth W and latency S costs

$$W_s = \Omega\left(\frac{|G|}{\sqrt{M}}\right), S_s = \Omega\left(\frac{|G|}{M^{3/2}}\right)$$

First noted by [Hong and Kung 81], this idea has been extended to a large class of computations [BDHS 11]

Parallel communication lower bounds

If **A**, **B**, and **C** are square n-by-n matrices, we have $|G| = n^3$, so

$$W_s = \Omega\left(\frac{|G|}{\sqrt{M}}\right) = \Omega\left(\frac{n^3}{\sqrt{M}}\right)$$

a similar lower bound argument holds during parallel execution with p processors [ITT 2004],

$$W_p = \Omega\left(\frac{n^3}{p \cdot \sqrt{M}}\right)$$

Parallel matrix multiplication algorithms

Standard '2D' algorithms ([Cannon 69], [GW 97], [ABGJP 95]) assume $M = 3n^2/p$ and block **A**, **B**, and **C**. They have a cost of

$$W_{2D} = O\left(\frac{n^2}{\sqrt{p}}\right)$$

'3D' algorithms ([Bernsten 89], [ACS 1990], [ABGJP 95], [MT 99]) assume $M = 3n^2/p^{2/3}$ and block the computation yielding

$$W_{3D} = O\left(\frac{n^2}{p^{2/3}}\right)$$

'2.5D' algorithms ([MT 99], [SD 2011]) generalize this and attain the lower bound for any $M \in [3n^2/p, 3n^2/p^{2/3}]$

$$W_{2.5D} = O\left(\frac{n^3}{p \cdot M}\right)$$

Tensor contractions

We define a tensor contraction between $\mathcal{A} \in \mathbb{R}^{\otimes k}$, $\mathcal{B} \in \mathbb{R}^{\otimes l}$ into $\mathcal{C} \in \mathbb{R}^{\otimes m}$ as

$$c_{i_1 i_2 \ldots i_m} = \sum_{j_1 j_2 \cdots j_{k+l-m}} a_{i_1 i_2 \ldots i_{m-l} j_1 j_2 \cdots j_{k+l-m}} \cdot b_{j_1 j_2 \cdots j_{k+l-m} i_{m-l+1} i_{m-l+2} \ldots i_m}$$

Tensor contractions reduce to matrix multiplication via index folding (let $[ijk]$ denote a group of 3 indices folded into one),

$$c_{[i_1 i_2 \ldots i_{m-l}],[i_{m-l+1} i_{m-l+2} \ldots i_m]} = \sum_{[j_1 j_2 \cdots j_{k+l-m}]} a_{[i_1 i_2 \ldots i_{m-l}],[j_1 j_2 \cdots j_{k+l-m}]} \cdot b_{[j_1 j_2 \cdots j_{k+l-m}],[i_{m-l+1} i_{m-l+2} \ldots i_m]}$$

so here $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ can be treated simply as matrices.

Communication lower bound for tensor contractions

The computational graph corresponding to a tensor contraction can be higher dimensional, but there are still only three projections corresponding to $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$. So, if the contraction necessitates F floating point operations, the bandwidth lower bound is still just

$$W_p = \Omega\left(\frac{F}{p \cdot \sqrt{M}}\right).$$

Therefore. folding contractions into matrix multiplication and running a good multiplication algorithm is communication-optimal.

Tensor symmetry

Tensors can have symmetry e.g.

$$a_{(ij)k} = a_{(ji)k} \quad \text{or} \quad a_{(ij)k} = -a_{(ji)k}$$

I am introducing more dubious notation, by denoting symmetric groups of indices as $(ab...)$. We now might face contractions like

$$c_{(ij)kl} = \sum_{pqr} a_{(ij)(pq)} \cdot b_{(pqk)(rl)}$$

where the computational graph G can be thought of as a 7D tensor with entries $g_{(ij)kl(pq)r} = (c_{(ij)kl}, a_{(ij)(pq)}, b_{(pqk)(rl)})$. There are two things that can happen to symmetries during a contraction:

- preserved, e.g. $g_{(ij)kl(pq)r} = g_{(ji)kl(pq)r}$
- broken, e.g. $b_{(pqk)(rl)} = b_{(pqk)(lr)}$ but $g_{(ij)kl(pq)r} \neq g_{(ij)kr(pq)l}$

Preserved symmetries in contractions

When a d-dimensional symmetry is preserved, a factor of $d!$ can be saved in memory and flops. This is simple to achieve, since the d-dimensional index group can be folded into one index in a packed layout, for instance

$$c_{kl} = 2 \cdot \sum_{[i<j]} a_{k[(i<j)]} \cdot b_{[(i<j)]l}$$

Since we are folding the packed index, the iteration space of this contraction is in effect equivalent to matrix multiplication, and therefore easy to handle.

Broken symmetries in contractions

When a symmetry is broken, no flops can be saved with respect to unpacking. However, memory can be saved as the tensors can remain stored in packed format. Matrix multiplication of two symmetric tensors features a broken symmetry, which can be computed in packed layout as

$$c_{kl} = \sum_i a_{(k<i)} \cdot b_{(i<l)} + a_{(i<k)} \cdot b_{(i<l)} + a_{(k<i)} \cdot b_{(l<i)} + a_{(i<k)} \cdot b_{(l<i)}$$

This requires four matrix multiplications, but each accesses only the lower triangle of the matrices, so only that portion need be stored.
If data replication is correctly utilized in the parallel algorithm unpacking and doing permutations of contractions have equivalent bandwidth costs.

NWChem approach to contractions

A high-level description of NWChem's algorithm for tensor contractions:

- data layout is abstracted away by the Global Arrays framework
- Global Arrays uses one-sided communication for data movement
- packed tensors are stored in blocks
- for each contraction, each process does a subset of the block contractions
- each block is transposed and unpacked prior to contraction
- automatic load balancing is employed among processors

Cyclops Tensor Framework (CTF) approach to contractions

A high-level description of CTF's algorithm for tensor contractions:

- tensor layout is carefully and dynamically orchestrated
- MPI collectives are used for all communication
- packed tensors are decomposed cyclically among processors
- for each contraction, a distributed layout is selected based on internal performance models
- before contraction, tensors are redistributed to a new layout
- if there is enough memory, the tensors are (partially) unpacked
- all preserved symmetries and non-symmetric indices are folded in preparation for GEMM
- nested distributed matrix multiply algorithms are used to perform the contraction in a load-balanced manner

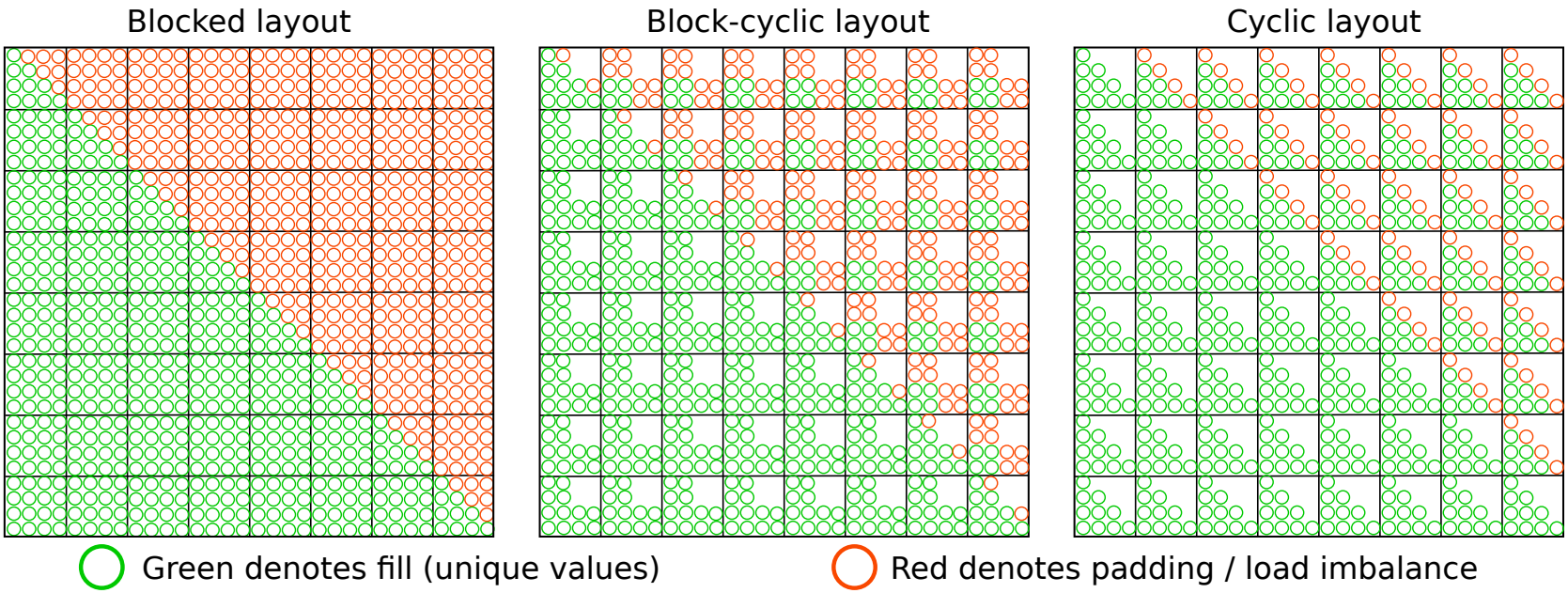
Cyclic decomposition in CTF

Cyclical distribution is fundamental to CTF, hence the name Cyclops (cyclic-operations).
Given a vector **v** of length n on p processors

- in a blocked distribution process p_i owns $\{v_{i\cdot n/p+1}, \ldots v_{(i+1)\cdot n/p}\}$
- in a cyclic distribution process p_i owns $\{v_i, v_{2i}, \ldots v_{(n/p)i}\}$

A cyclic distribution is associated with a phase along each dimension (for the vector above this was p). The main advantage from this distribution is that each subtensor can retain packed structure with only minimal padding.
CTF assumes all subtensor symmetries have index relations of the form $\leq$ and not $<$, so in effect, diagonals are stored for skew-symmetric tensors.

Blocked vs block-cyclic vs cyclic decompositions



Sequential tensor contractions

A cyclic distribution provides a vital level of abstraction, because each subtensor contraction becomes a packed contraction of the same sort as the global tensor contraction but of smaller size. Given a sequential packed contraction kernel, CTF can parallelize it automatically. Further, because each subcontraction is the same, the workload of each processor is the same. The actual sequential kernel used by CTF employs the following steps

1. if there is enough memory, unpack broken symmetries
2. perform a nonsymmetric transpose, to make all indices of non-broken symmetry be the leading dimensions
3. use a naive kernel to iterate though indices with broken symmetry and call BLAS GEMM for the leading dimensions

A cyclic layout is still challenging

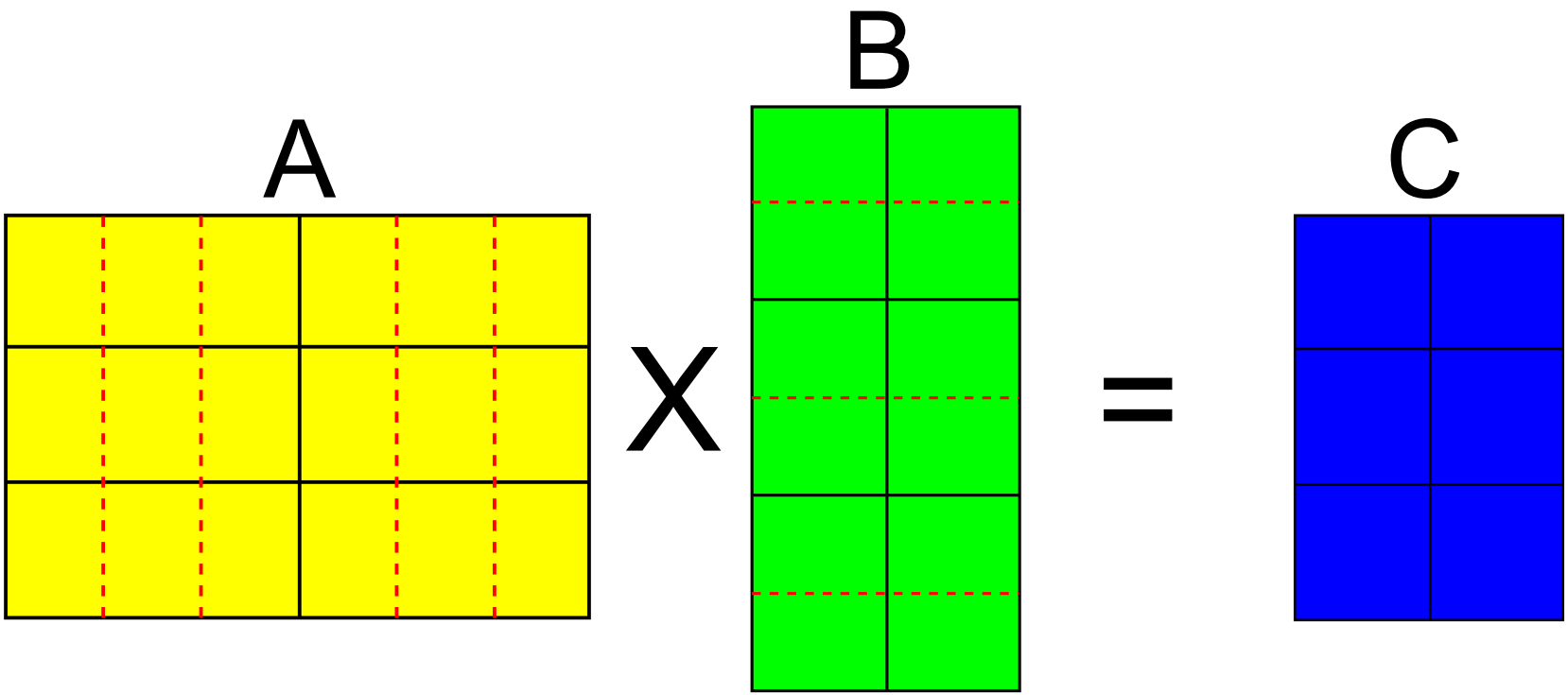
- In order to retain structure, all symmetric dimensions of a tensor must be mapped with the same cyclic phase
- The contracted dimensions of A and B must be mapped with the same phase
- This logical mapping still needs to be mapped to a physical network topology, which can be any shape

Virtual processor grid dimensions

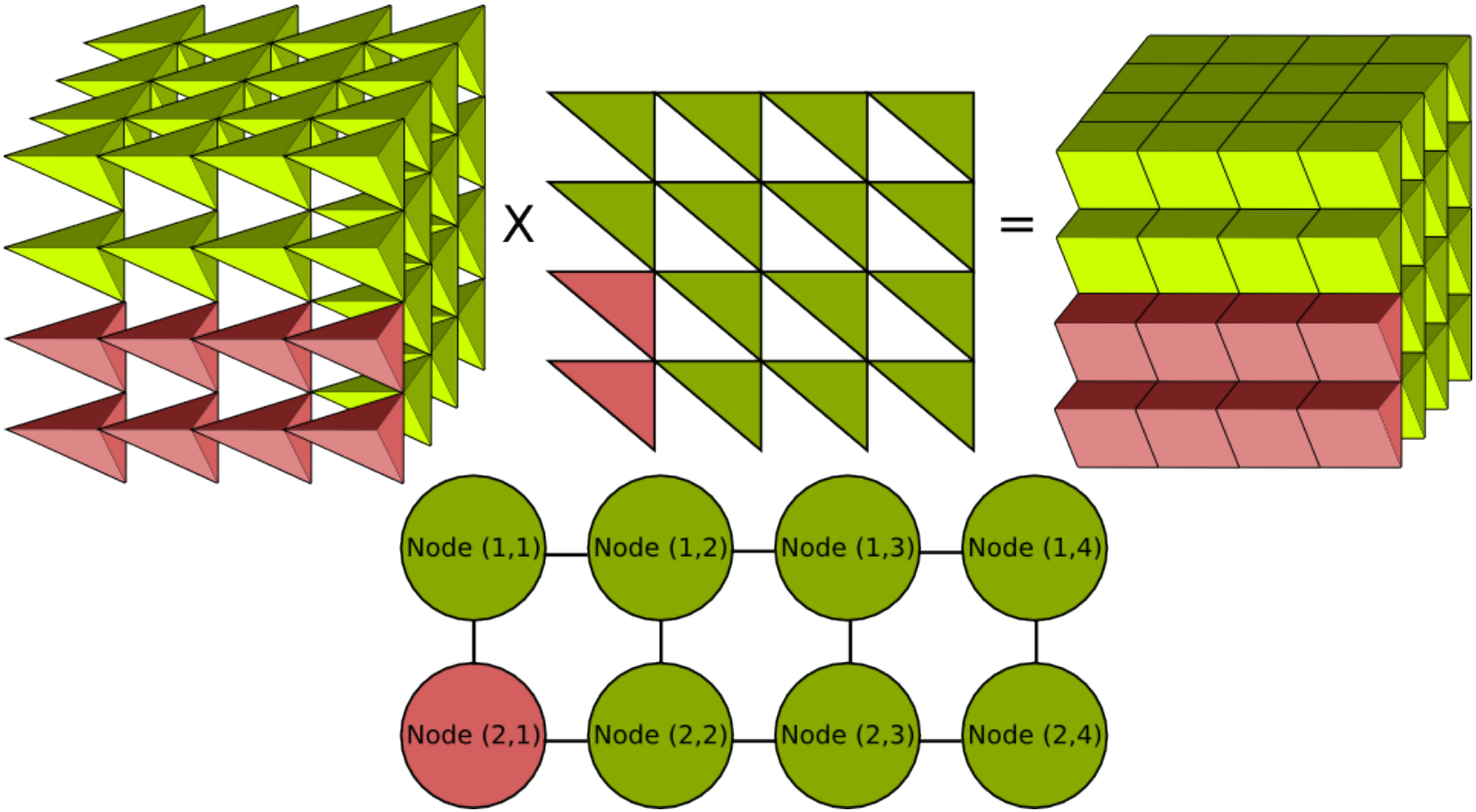
- Our virtual cyclic topology is somewhat restrictive and the physical topology is very restricted
- Virtual processor grid dimensions serve as a new level of indirection
  - If a tensor dimension must have a certain cyclic phase, adjust physical mapping by creating a virtual processor dimension
  - Allows physical processor grid to be 'stretchable'

Virtual processor grid construction

Matrix multiply on 2x3 processor grid. Red lines represent virtualized part of processor grid. Elements assigned to blocks by cyclic phase.



3D tensor mapping



Multidimensional processor grids

CTF supports tensors and processor grids of any dimension because mapping a symmetric tensor to a processor grid of the same dimension preserves symmetric structure with minimal virtualization and padding. Processor grids are defined by

- a base grid, obtained from the physical topology or from factorizing the number of processors
- folding all possible combinations of adjacent processor grid dimensions

Tensors are contracted on higher dimensional processor grids by

- mapping an index shared by two tensors in the contraction to different processor grid dimensions
- running a distributed matrix multiplication algorithm for each such 'mismatched' index
- replicating data along some processor dimensions 'a la 2.5D'

2.5D algorithms for tensors

We incorporate data replication for communication minimization into CTF

- Replicate only one tensor/matrix (minimize bandwidth but not latency)
- In parallel, autotune over mappings to all possible physical topologies
- Select mapping with least amount of communication that fits in memory
- Achieve minimal communication for tensors of widely different sizes

Tensor redistribution

Between each contraction, a new layout is typically selected for each tensor. This is a serious challenge, because a number of things change at once

- the processor grid changes
- the virtualization factors change
- the padding along each dimension changes

A lot of effort has been put into making redistributions fast in CTF, including a complex linear-time kernel and threading thereof. While this kernel can still consume a significant fraction of the time, it ceases to be a bottleneck for large contractions and scales well.

Coupled Cluster definition

Coupled Cluster (CC) is a method for computing an approximate solution to the time-independent Schrödinger equation of the form

$$\mathbf{H}|\Psi\rangle = E|\Psi\rangle,$$

CC rewrites the wave-function $|\Psi\rangle$ as an excitation operator $\hat{\mathbf{T}}$ applied to the Slater determinant $|\Phi_0\rangle$

$$|\Psi\rangle = e^{\hat{\mathbf{T}}}|\Phi_0\rangle$$

where $\hat{\mathbf{T}}$ is as a sum of $\hat{\mathbf{T}}_n$ (the n'th excitation operators)

$$\hat{\mathbf{T}}_{\mathrm{CCSD}} = \hat{\mathbf{T}}_1 + \hat{\mathbf{T}}_2$$
$$\hat{\mathbf{T}}_{\mathrm{CCSDT}} = \hat{\mathbf{T}}_1 + \hat{\mathbf{T}}_2 + \hat{\mathbf{T}}_3$$
$$\hat{\mathbf{T}}_{\mathrm{CCSDTQ}} = \hat{\mathbf{T}}_1 + \hat{\mathbf{T}}_2 + \hat{\mathbf{T}}_3 + \hat{\mathbf{T}}_4$$

Coupled Cluster derivation

To derive CC equations, a normal-ordered Hamiltonian is defined as the sum of one-particle and two-particle interaction terms

$$\hat{\mathbf{H}}_N = \hat{\mathbf{F}}_N + \hat{\mathbf{V}}_N$$

Solving the CC energy contribution can be done by computing eigenvectors of the similarity-transformed Hamiltonian

$$\bar{\mathbf{H}} = e^{-\hat{\mathbf{T}}} \hat{\mathbf{H}}_N e^{\hat{\mathbf{T}}}$$

Performing the CCSD truncation $\hat{\mathbf{T}} = \hat{\mathbf{T}}_1 + \hat{\mathbf{T}}_2$ and applying the Hadamard lemma of the Campbell-Baker-Hausdorff formula,

$$\bar{\mathbf{H}} = \hat{\mathbf{H}}_N + [\hat{\mathbf{H}}_N, \hat{\mathbf{T}}_1] + [\hat{\mathbf{H}}_N, \hat{\mathbf{T}}_2] + \frac{1}{2}[[\hat{\mathbf{H}}_N \hat{\mathbf{T}}_1], \hat{\mathbf{T}}_1] \ldots$$

which simplifies to

$$\bar{\mathbf{H}} = \hat{\mathbf{H}}_N + \hat{\mathbf{H}}_N \hat{\mathbf{T}}_1 + \hat{\mathbf{H}}_N \hat{\mathbf{T}}_2 + \hat{\mathbf{H}}_N \hat{\mathbf{T}}_1^2 + \ldots$$

Coupled Cluster equations

Left projecting the eigenvector equation, we can obtain an explicit formula for the CC energy via Wick contraction

$$E_{\text{CCSD}} - E_0 = \langle \Phi_0 | \bar{\mathbf{H}} | \Phi_0 \rangle = \sum_{ia} f_{ia} t_i^a + \frac{1}{4} \sum_{abij} \langle ij||ab \rangle t_{ij}^{ab} + \frac{1}{2} \sum_{aibj} \langle ij||ab \rangle t_i^a t_j^b$$

The tensor amplitude equations are derived in a similar fashion but involve many more terms

$$0 = \langle \Phi_i^a | \bar{\mathbf{H}} | \Phi_0 \rangle = f_{ai} - \sum_{kc} f_{kc} t_i^c t_k^a + \dots$$

$$0 = \langle \Phi_{ij}^{ab} | \bar{\mathbf{H}} | \Phi_0 \rangle = \langle ab||ij \rangle + \sum_{bj} \langle ja||bi \rangle t_j^b + \dots$$

These equations then need to be factorized into two-tensor contractions.

Spin symmetry in tensors

- CC methods all deal with the tensors $\hat{\mathbf{F}}$, $\hat{\mathbf{V}}$, $\hat{\mathbf{T}}_n$. In the spin orbital formulation, these tensors have skew-symmetry e.g. $\langle ab||ij\rangle = v_{(ij)}^{(ab)}$ and $t_{(ijk)}^{(abc)}$. EOM methods introduce an excitation operator $\hat{\mathbf{R}}$, which has similar structure.
- The tensors can be further factored into spin-blocks and the resulting spin-integrated equations can factor out mixed-spin terms. This yields a subdivision of previously symmetric tensors into blocks, some of which are symmetric and some of which are nonsymmetric.
- Some molecules also have point-group symmetries, which allow factorization of the two-electron and amplitude tensors into more blocks and restricting the equations further.

Interfacing tensor contractions

At user-level, we aim to hide all the complexity of the parallel framework and allow simple specification of contractions. The interface uses Einstein notation and allows iteration over diagonals

$$\mathcal{C}[''abij'']+=\mathcal{A}[''acij'']\cdot\mathcal{B}[''cb'']$$

$$\mathcal{C}[''aj'']+=\mathcal{A}[''aaaj'']\cdot\mathcal{B}[''bb'']$$

with symmetries and spin-cases pre-specified for A, B, and C.

Tensor definition

We allow the user to specify the spin-cases for each tensor and perform the necessary contractions automatically. It seems natural that this approach should be extensible to point-group symmetry

```
SpinorbitalTensor<DistTensor> T2("ab,ij");
T2.addSpinCase(new DistTensor(4, sizeVVOO, shapeANAN, dw), "AB,IJ", "ABIJ");
T2.addSpinCase(new DistTensor(4, sizeVvOo, shapeNNNN, dw), "Ab,Ij", "AbIj");
T2.addSpinCase(new DistTensor(4, sizevvoo, shapeANAN, dw), "ab,ij", "abij");
```

CCSD

- UHF
- spin-integrated
- does not use disk
- does not exploit point-group symmetry
- not quite fully mature SCF, integrals, and IO, currently used only for testing

Our CCSD factorization

Credit to John F. Stanton and Jurgen Gauss

$$\tau_{ij}^{ab} = t_{ij}^{ab} + \frac{1}{2} P_b^a P_j^i t_i^a t_j^b,$$

$$\tilde{F}_e^m = f_e^m + \sum_{fn} v_{ef}^{mn} t_n^f,$$

$$\tilde{F}_e^a = (1-\delta_{ae}) f_e^a - \sum_m \tilde{F}_e^m t_m^a - \frac{1}{2} \sum_{mnf} v_{ef}^{mn} t_{mn}^{af} + \sum_{fn} v_{ef}^{an} t_n^f,$$

$$\tilde{F}_i^m = (1-\delta_{mi}) f_i^m + \sum_e \tilde{F}_e^m t_i^e + \frac{1}{2} \sum_{nef} v_{ef}^{mn} t_{in}^{ef} + \sum_{fn} v_{if}^{mn} t_n^f,$$

Our CCSD factorization

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

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

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

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

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

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

Actual CCSD code

```
FVO["me"] = VABIJ["efmn"]*T1["fn"];
FVV["ae"] = -0.5*VABIJ["femn"]*T2["famn"];
FVV["ae"] -= FVO["me"]*T1["am"];
FVV["ae"] += VABCI["efan"]*T1["fn"];
FOO["mi"] = 0.5*VABIJ["efnm"]*T2["efni"];
FOO["mi"] += FVO["me"]*T1["ei"];
FOO["mi"] += VIJKA["mnif"]*T1["fn"];
WMNIJ["mnij"] = VIJKL["mnij"];
WMNIJ["mnij"] += 0.5*VABIJ["efmn"]*Tau["efij"];
WMNIJ["mnij"] += VIJKA["mnie"]*T1["ej"];
WMNIE["mnie"] = VIJKA["mnie"];
WMNIE["mnie"] += VABIJ["femn"]*T1["fi"];
WAMIJ["amij"] = VIJKA["jima"];
WAMIJ["amij"] += 0.5*VABCI["efam"]*Tau["efij"];
WAMIJ["amij"] += VAIBJ["amej"]*T1["ei"];
WMAEI["maei"] = -VAIBJ["amei"];
WMAEI["maei"] += 0.5*VABIJ["efmn"]*T2["afin"];
WMAEI["maei"] += VABCI["feam"]*T1["fi"];
WMAEI["maei"] -= WMNIE["nmie"]*T1["an"];
Z1["ai"] = 0.5*VABCI["efam"]*Tau["efim"];
Z1["ai"] -= 0.5*WMNIE["mnie"]*T2["aemn"];
Z1["ai"] += T2["aeim"]*FVO["me"];
Z1["ai"] -= T1["em"]*VAIBJ["amei"];
Z1["ai"] -= T1["am"]*FOO["mi"];
Z2["abij"] = VABIJ["abij"];
Z2["abij"] += FVV["af"]*T2["fbij"];
Z2["abij"] -= FOO["ni"]*T2["abnj"];
Z2["abij"] += VABCI["abej"]*T1["ei"];
Z2["abij"] -= WAMIJ["mbij"]*T1["am"];
Z2["abij"] += 0.5*VABCD["abef"]*Tau["efij"];
Z2["abij"] += 0.5*WMNIJ["mnij"]*Tau["abmn"];
Z2["abij"] += WMAEI["maei"]*T2["ebmj"];
E1["ai"]    = Z1["ai"]  *D1["ai"];
E2["abij"]  = Z2["abij"]*D2["abij"];
E1["ai"]   -= T1["ai"];
E2["abij"] -= T2["abij"];
T1["ai"]   += E1["ai"];
T2["abij"] += E2["abij"];

Tau["abij"]  = T2["abij"];
Tau["abij"] += 0.5*T1["ai"]*T1["bj"];

E_CCSD = 0.25*scalar(VABIJ["efmn"]*Tau["efmn"]);
```

Sequential performance comparison

The sequential performance is subpar, surprisingly due to the nonsymmetric transpose (especially when the number of occupied orbitals is relatively small). CCSD performance on a Xeon E5620, single threaded, Intel MKL.

	H2O/cc-pVQZ	C3H4/aug-cc-pVDZ	C2H2O4/cc-pVDZ
electrons	5	11	23
orbitals	115	105	94
MRCC	31 s/iter	66.2 s/iter	224 s/iter
NWChem	6.8 s/iter	16.8 s/iter	49 s/iter
CFOUR	4.9 s/iter	13 s/iter	34.7 s/iter
CTF	23.6 s/iter	32.5 s/iter	59.8 s/iter

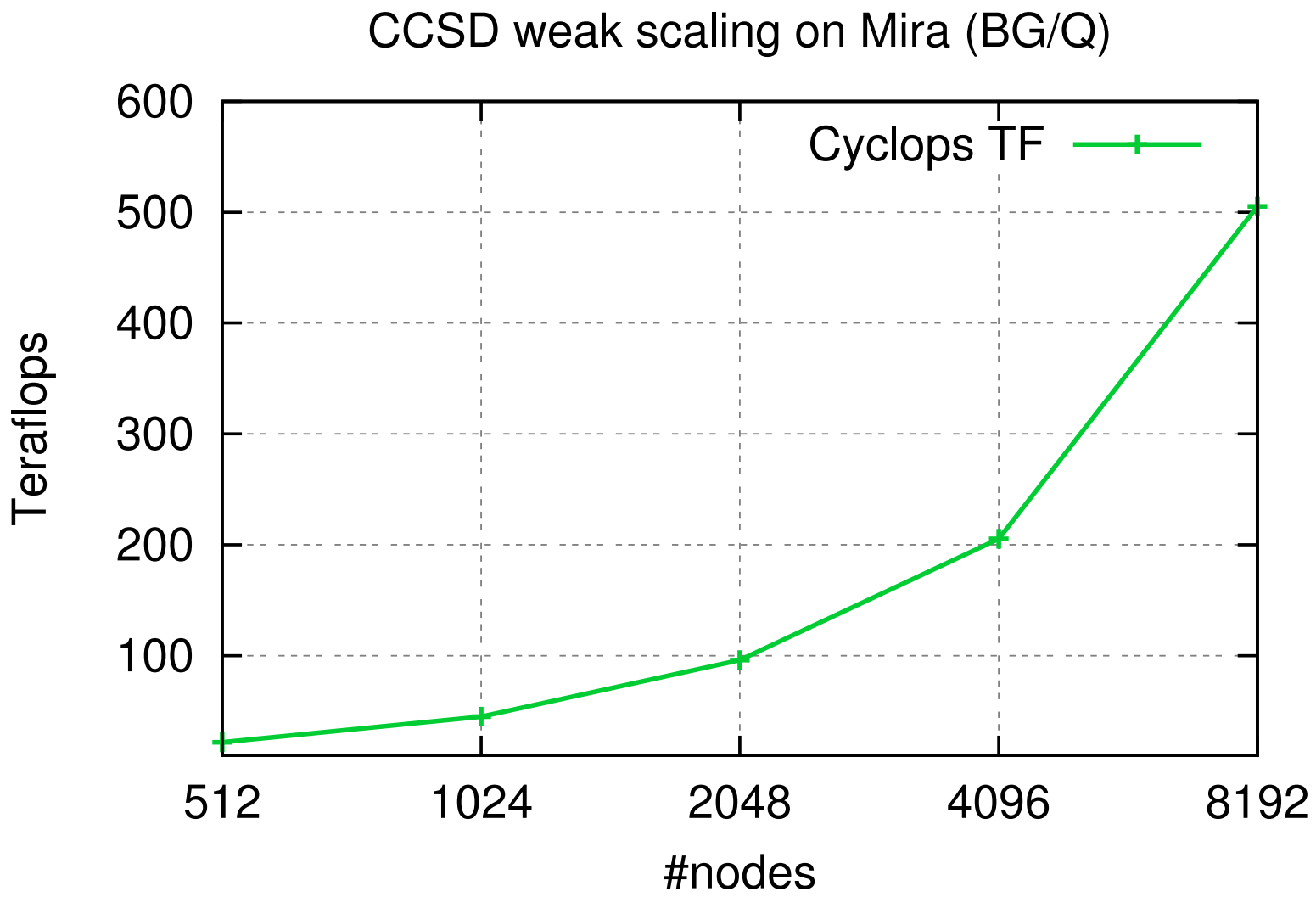
Comparison with NWChem on Cray XE6

CCSD iteration time on 64 nodes of Hopper:

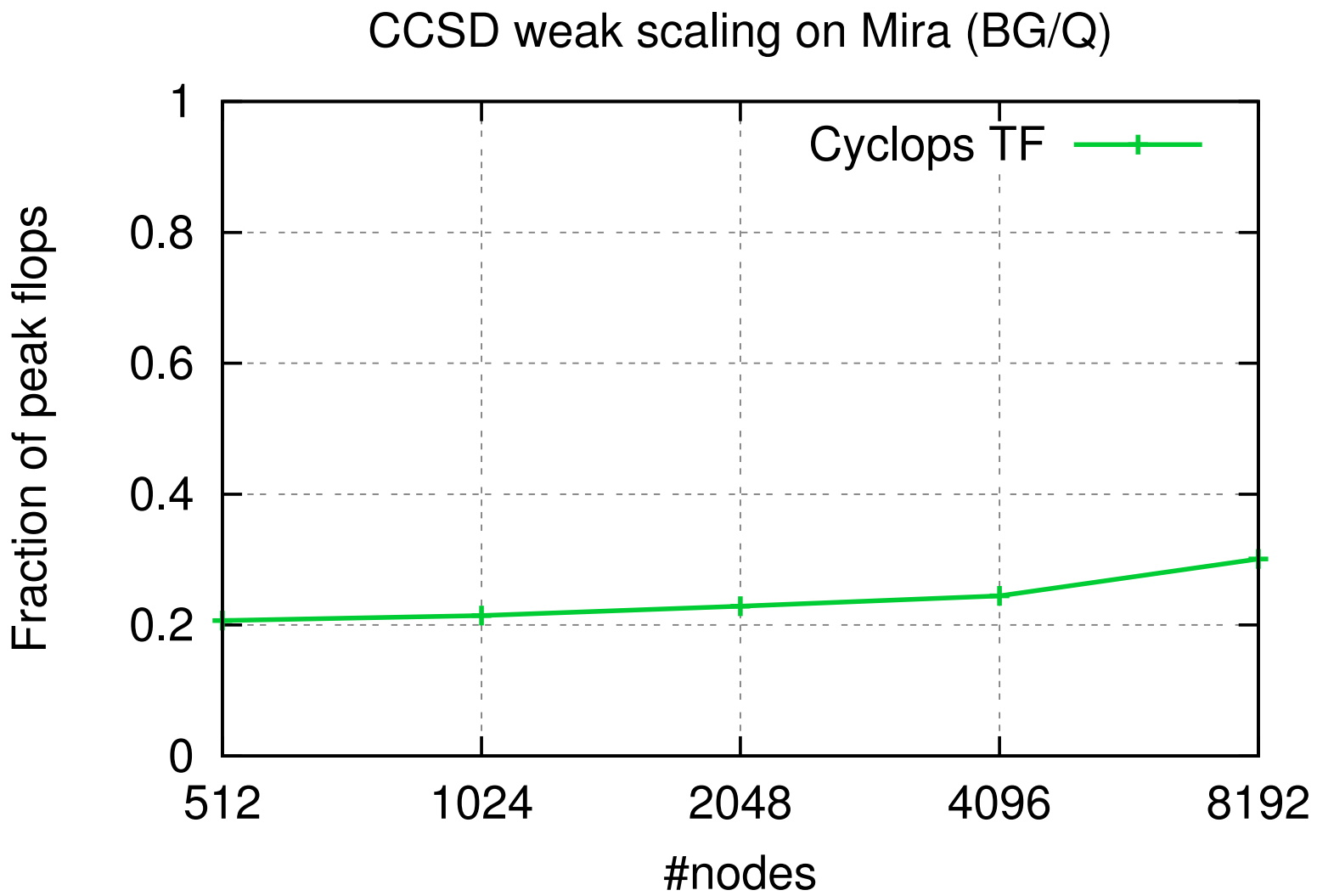
system	# electrons	# orbitals	CTF	NWChem
w5	25	205	14 sec	36 sec
w7	35	287	90 sec	178 sec
w9	45	369	127 sec	-
w12	60	492	336 sec	-

On 128 nodes, NWChem completed w9 in 223 sec, CTF in 73 sec.

Blue Gene/Q up to 1250 orbitals, 250 electrons



Coupled Cluster efficiency on Blue Gene/Q



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

Performance breakdown on Cray XE6

Performance data for a CCSD iteration with 100 electrons and 500 orbitals on 256 nodes of Hopper

4 processes per node, 6 threads per process

Total time: 9 mins

v-orbitals, o-electrons

kernel	% of time	complexity	architectural bounds
DGEMM	21% $\Downarrow$ 24%	$O(v^4o^2/p)$	flops/mem bandwidth
broadcasts	32% $\Uparrow$ 12%	$O(v^4o^2/p\sqrt{M})$	multicast bandwidth
prefix sum	7% $\Downarrow$ 3%	$O(p)$	allreduce bandwidth
data packing	10% $\Uparrow$ 3%	$O(v^2o^2/p)$	integer ops
all-to-all-v	8%	$O(v^2o^2/p)$	bisection bandwidth
tensor folding	4%	$O(v^2o^2/p)$	memory bandwidth

Higher-order Coupled Cluster

- CTF and surrounding infrastructure has been implemented with no assumptions on maximum tensor dimension or symmetry group size (does not mean there are no bugs involving higher dimensions).
- CCSDT is in the works
- CCSDTQ is next on the plan
- CCSD(T) is currently lower priority because it will necessitate more work

Sparsity

- Coupled Cluster does not scale with system size!
- there is promising theory around associating a subset of virtual orbitals with each electron
- there is a variety of low-rank decomposition methods, though they seem orthogonal to this work
- multi-scale methods? (glue can be harder than the original problem)

An inquiry

Is there any application for symmetric tensor operations other than contractions, e.g. eigenvalue problem or linear solver? For example, solve for x in

$$a_{(ijk)} \cdot x_{kl} = b_{(ij)l}$$

such operations are well-defined because they are mapped to matrices, and seem to pose interesting algorithmic questions.

Summary and conclusion

- Communication cost and load balance matter, especially in parallel
- Tensor contractions reduce to matrix multiplication, but symmetries yield complications
- CTF resolves some of the challenges in a novel way, but has its own drawbacks
- CTF runs in parallel

Backup slides

Density Function Theory (DFT)

DFT uses the fact that the ground-state wave-function Ψ_0 is a unique functional of the particle density $n(\vec{r})$

$$\Psi_0 = \Psi[n_0]$$

Since $\hat{H} = \hat{T} + \hat{V} + \hat{U}$, where $\hat{T}$, $\hat{V}$, and $\hat{U}$, are the kinetic, potential, and interaction contributions respectively,

$$E[n_0] = \langle \Psi[n_0] | \hat{T}[n_0] + \hat{V}[n_0] + \hat{U}[n_0] | \Psi[n_0] \rangle$$

DFT assumes $\hat{U} = 0$, and solves the Kohn-Sham equations

$$\left[-\frac{\hbar^2}{2m} \nabla^2 + V_s(\vec{r}) \right] \phi_i(\vec{r}) = \epsilon_i \phi_i(\vec{r})$$

where V_s has a exchange-correlation potential correction,

$$V_s(\vec{r}) = V(\vec{r}) + \int \frac{e^2 n_s(\vec{r'})}{|\vec{r} - \vec{r'}|} d^3 r' + V_{XC}[n_s(\vec{r})]$$

Density Function Theory (DFT), contd.

The exchange-correlation potential V_{XC} is approximated by DFT, by a functional which is often system-dependent. This allows the following iterative scheme

1. Given an (initial guess) $n(\vec{r})$ calculate V_s via Hartree-Fock and functional
2. Solve (diagonalize) the Kohn-Sham equation to obtain each ϕ_i
3. Compute a new guess at $n(\vec{r})$ based on ϕ_i

Due to the rough approximation of correlation and exchange DFT is good for weakly-correlated systems (which appear in solid-state physics), but suboptimal for strongly-correlated systems.

Linear algebra in DFT

DFT requires a few core numerical linear algebra kernels

- Matrix multiplication (of rectangular matrices)
- Linear equations solver
- Symmetric eigensolver (diagonalization)

We proceed to study schemes for optimization of these algorithms.

Solutions to linear systems of equations

We want to solve some matrix equation

$$A \cdot X = B$$

where A and B are known. Can solve by factorizing $A = LU$ (L lower triangular and U upper triangular) via Gaussian elimination, then computing TRSMs

$$X = U^{-1}L^{-1}B$$

via triangular solves. If A is symmetric positive definite, we can use Cholesky factorization. Cholesky and TRSM are no harder than LU.

Non-pivoted LU factorization

for $k = 0$ to $n-1$ **do**
  $U[k, k:n-1] = A[k, k:n-1]$
  for $i = k+1$ to $n-1$ **do**
    $L[i,k] = A[i,k]/U[k,k]$
    for $j = k+1$ to $n-1$ **do**
      $A[i,j] -= L[i,k] \cdot U[k,j]$

This algorithm has a dependency that requires

$$k \leq i, k \leq j.$$

Non-pivoted 2D LU factorization

On a l-by-l process grid

Algorithm 1 $[L, U] = \text{2D-LU}(A)$

for $k = 0$ to $n-1$ **do**
  Factorize $A[k,k] = L[k,k] \cdot U[k,k]$
  Broadcast $L[k,k]$ and $U[k,k]$
  for $p = 0$ to $l-1$ **in parallel do**
    solve $L[p,k] = A[p,k]U[k,k]^{-1}$
  for $q = 0$ to $l-1$ **in parallel do**
    solve $U[k,q] = L[k,k]^{-1}A[1,k]$
  Broadcast $L[p,k]$ and $U[k,q]$
  for $p = 0$ to $l-1$ **in parallel do**
    for $q = 0$ to $l-1$ **in parallel do**
      $A[p,q] -= L[p,k] \cdot U[k,q]$

3D recursive non-pivoted LU and Cholesky

A 3D recursive algorithm with no pivoting [A. Tiskin 2002]

- Tiskin gives algorithm under the BSP model
  - Bulk Synchronous Parallel
  - considers communication and synchronization
- We give an alternative distributed-memory adaptation and implementation
- Also, we have a new lower-bound for the latency cost

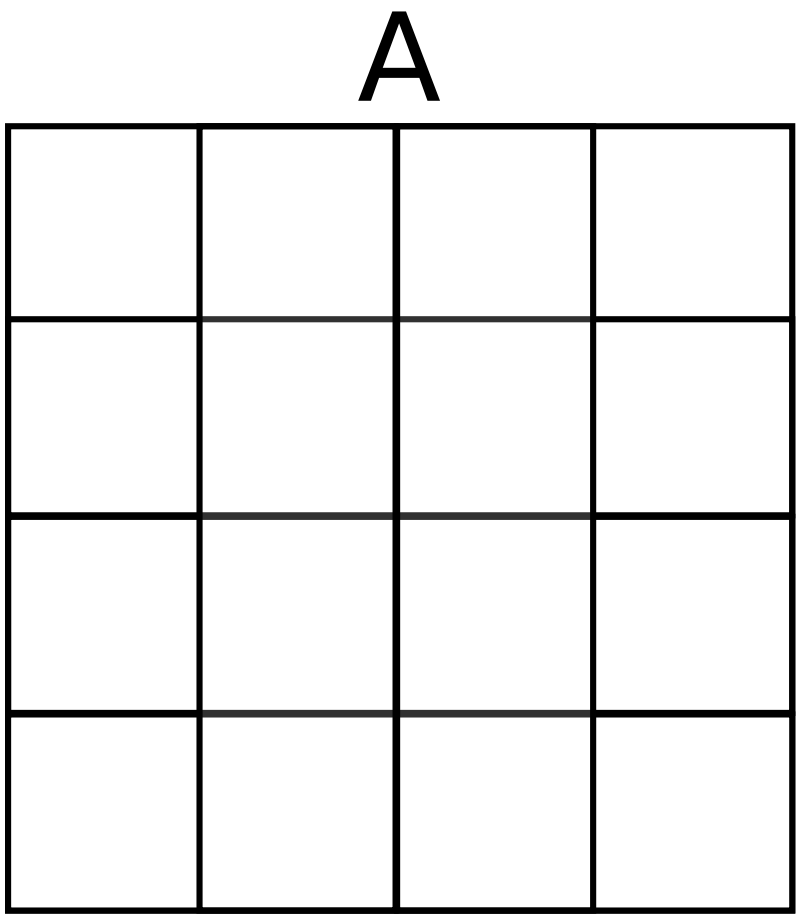
3D non-pivoted LU and Cholesky

On a l-by-l-by-l process grid

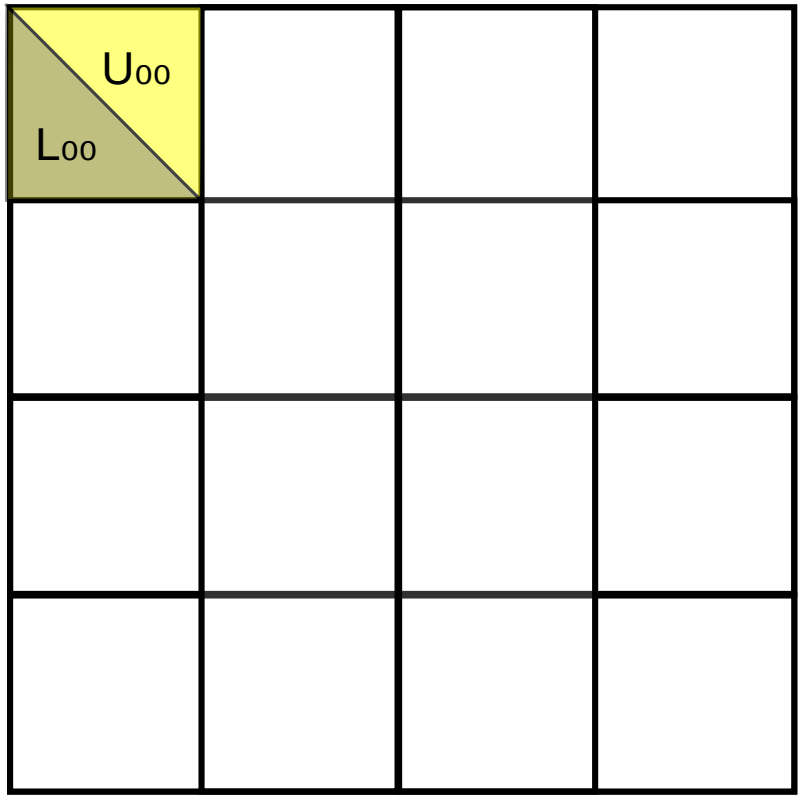
---

for $r = 0$ to $l-1$ **do**
  $[L[r,r], U[r,r]] = \text{2D-LU}(A[r,r])$
  Broadcast $L[k,k]$ and $U[k,k]$
  $[L[r+1:l-1, r]] = \text{2D-TRSM}(A[r+1:l-1, r], U[r,r])$;
  $[U[r, r+1:l-1]] = \text{2D-TRSM}(A[r, r+1:l-1], L[r,r])$;
  for $s = 0$ to $l-1$ **in parallel do**
    Broadcast $L[p, rs]$ and $U[rs, q]$
    for $p = 0$ to $l-1$ **in parallel do**
      for $q = 0$ to $l-1$ **in parallel do**
        $A[p,q] -= L[p,rs] \cdot U[rs,q]$

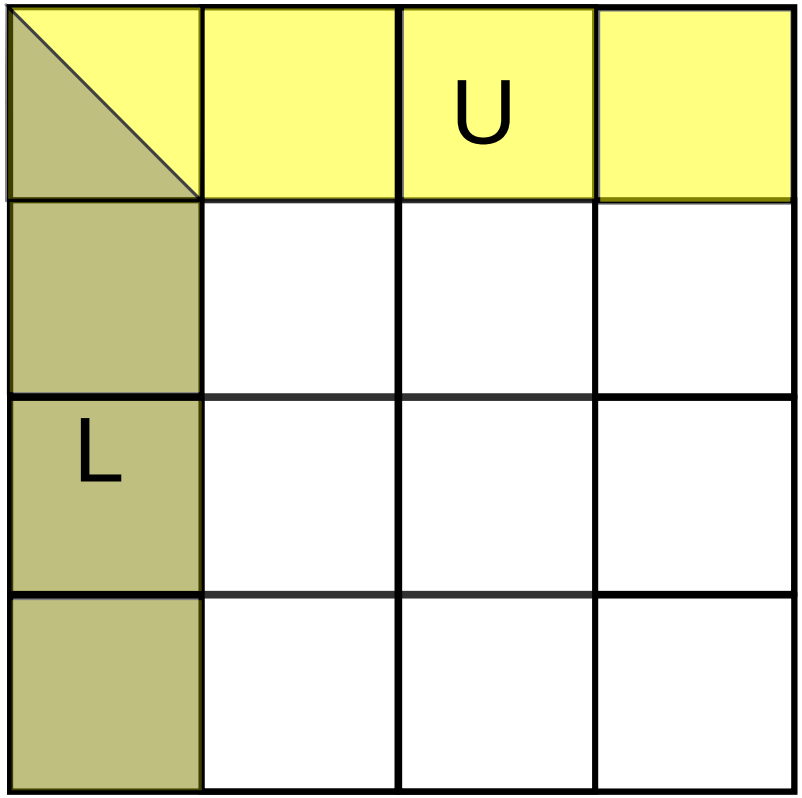
2D blocked LU factorization



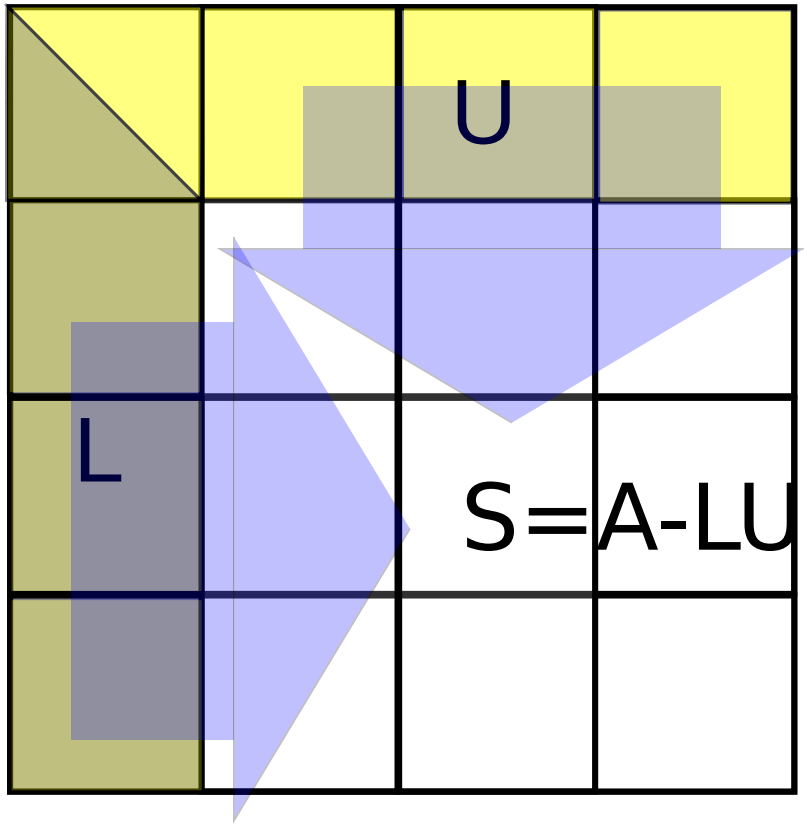
2D blocked LU factorization



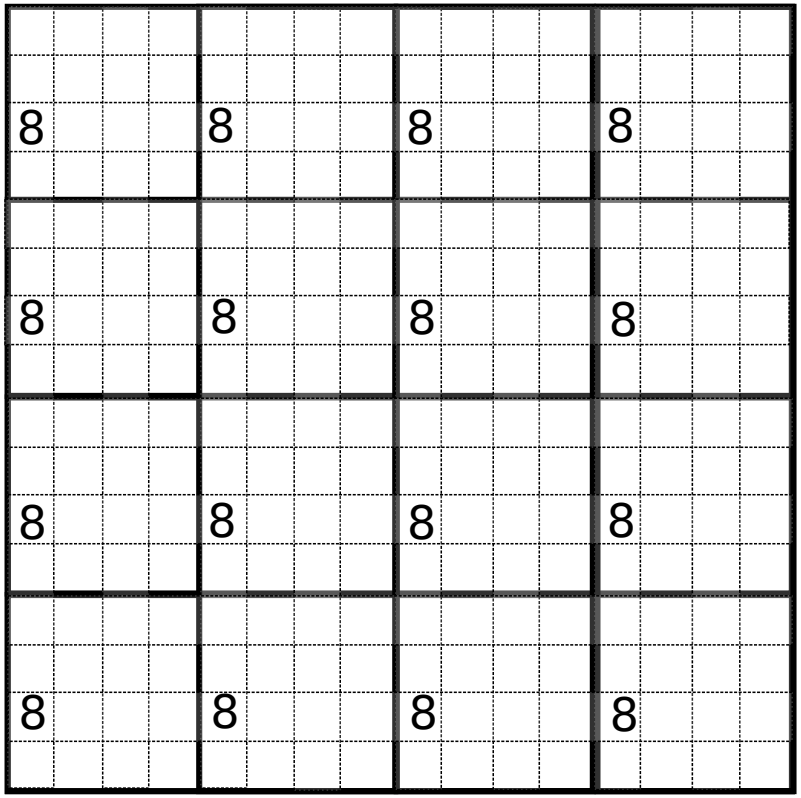
2D blocked LU factorization



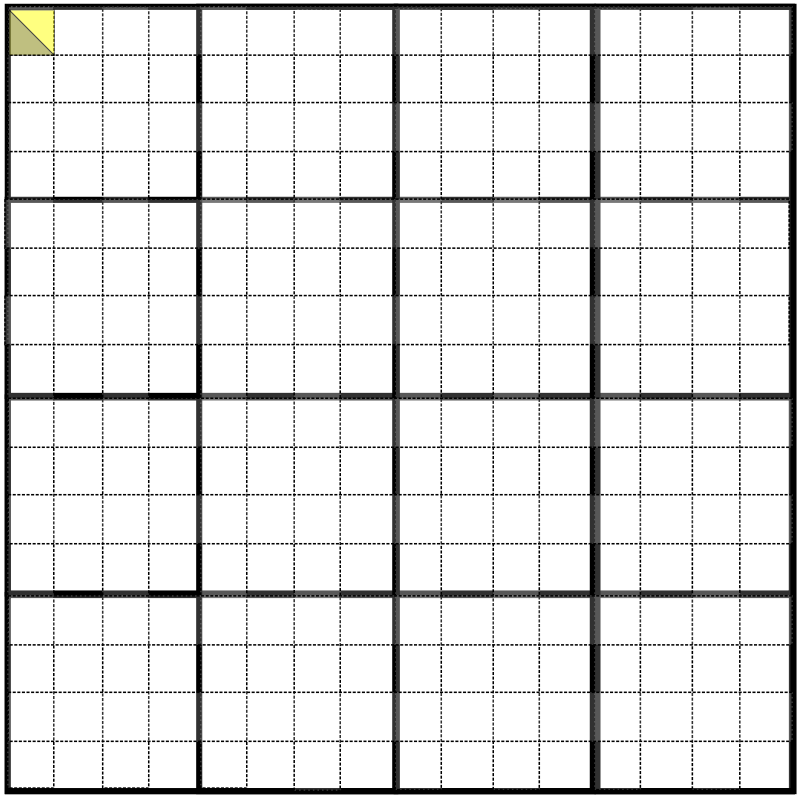
2D blocked LU factorization



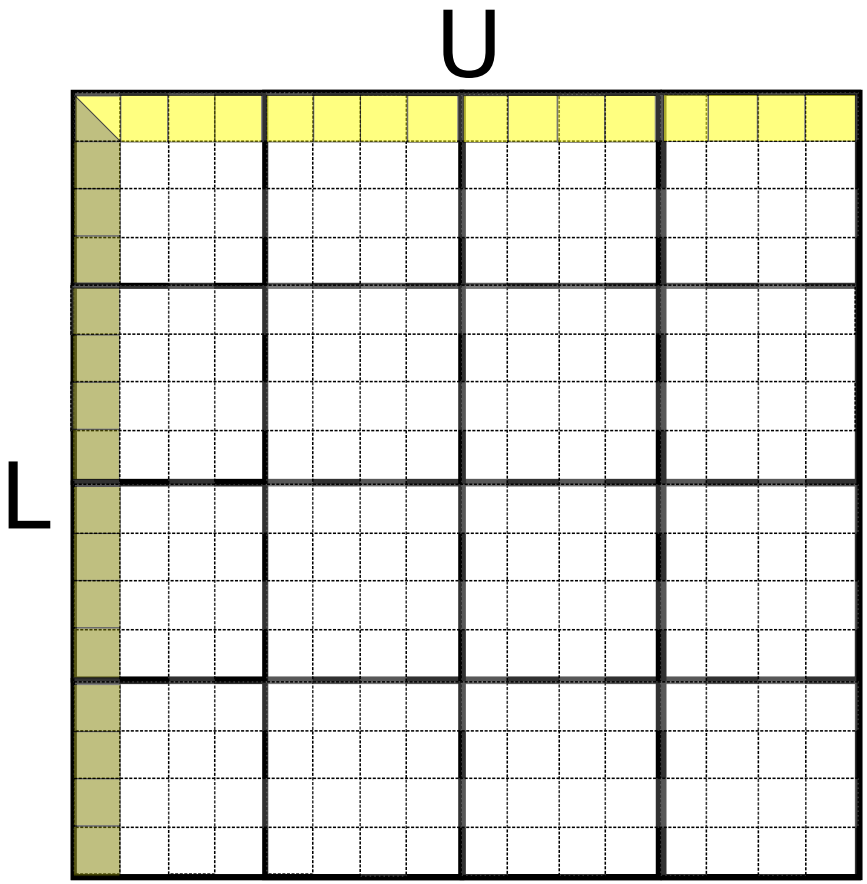
2D block-cyclic decomposition



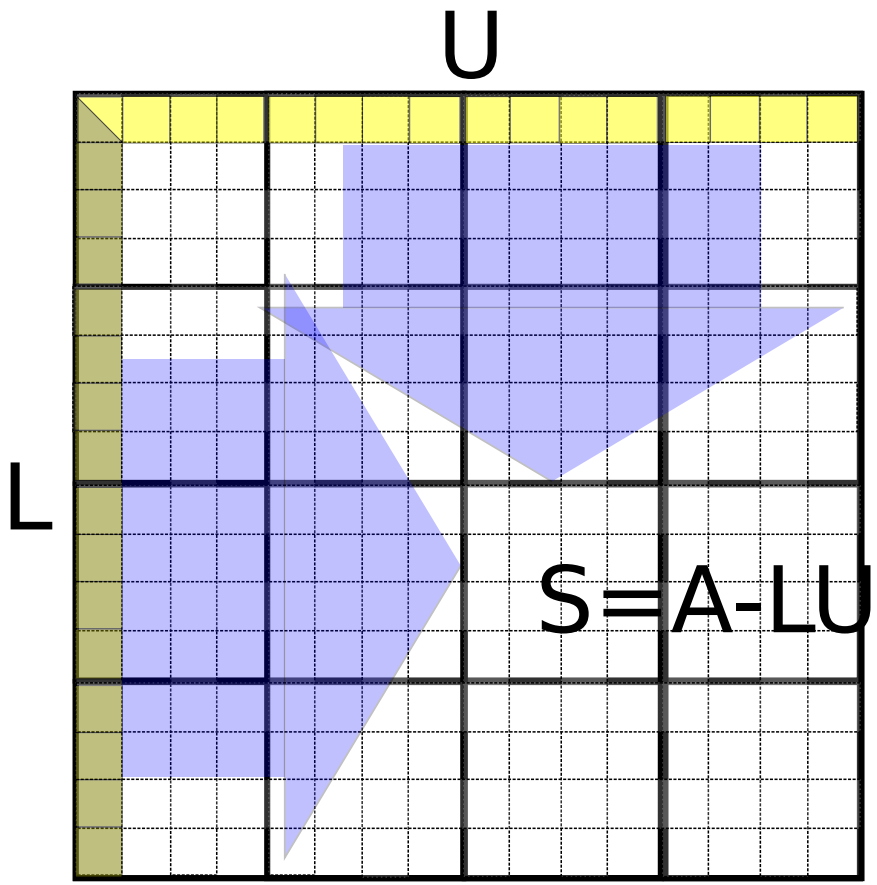
2D block-cyclic LU factorization



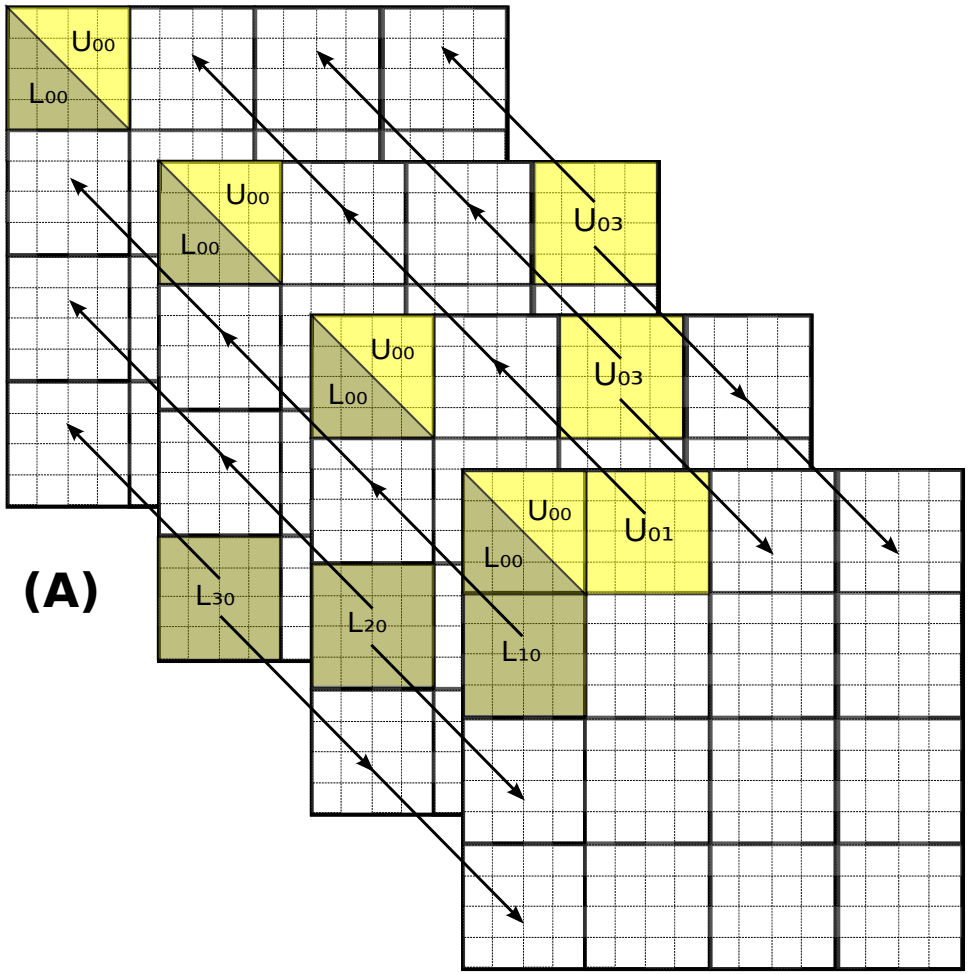
2D block-cyclic LU factorization



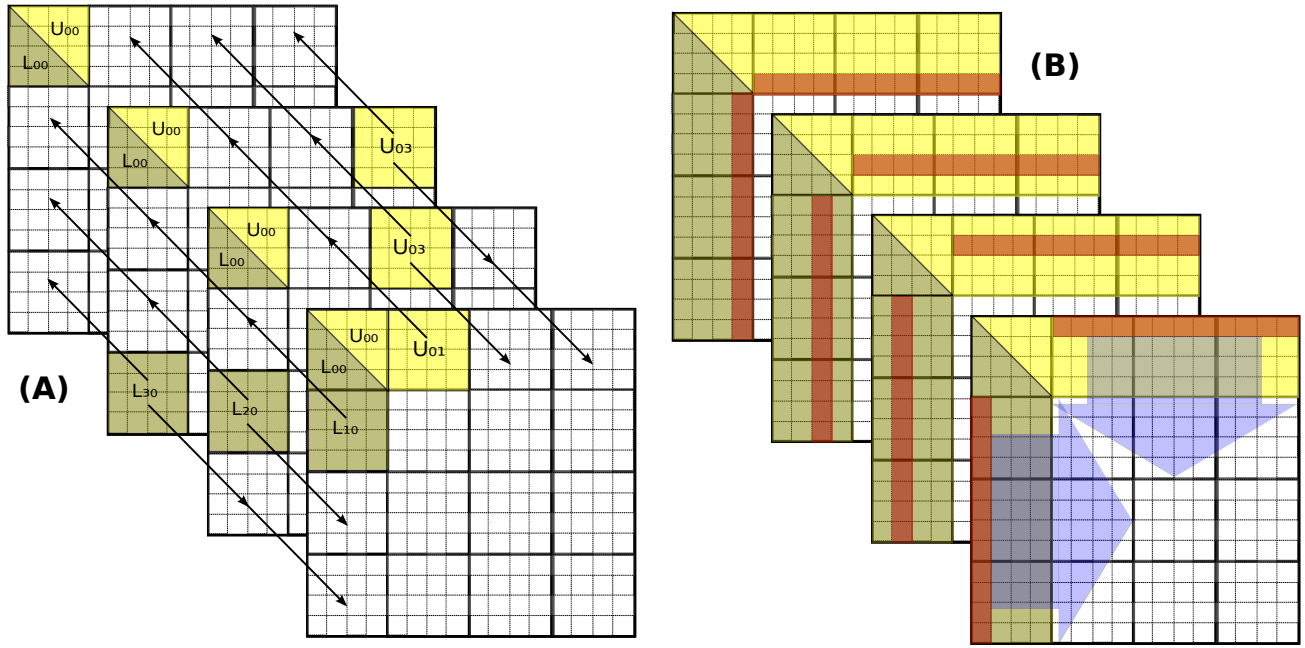
2D block-cyclic LU factorization



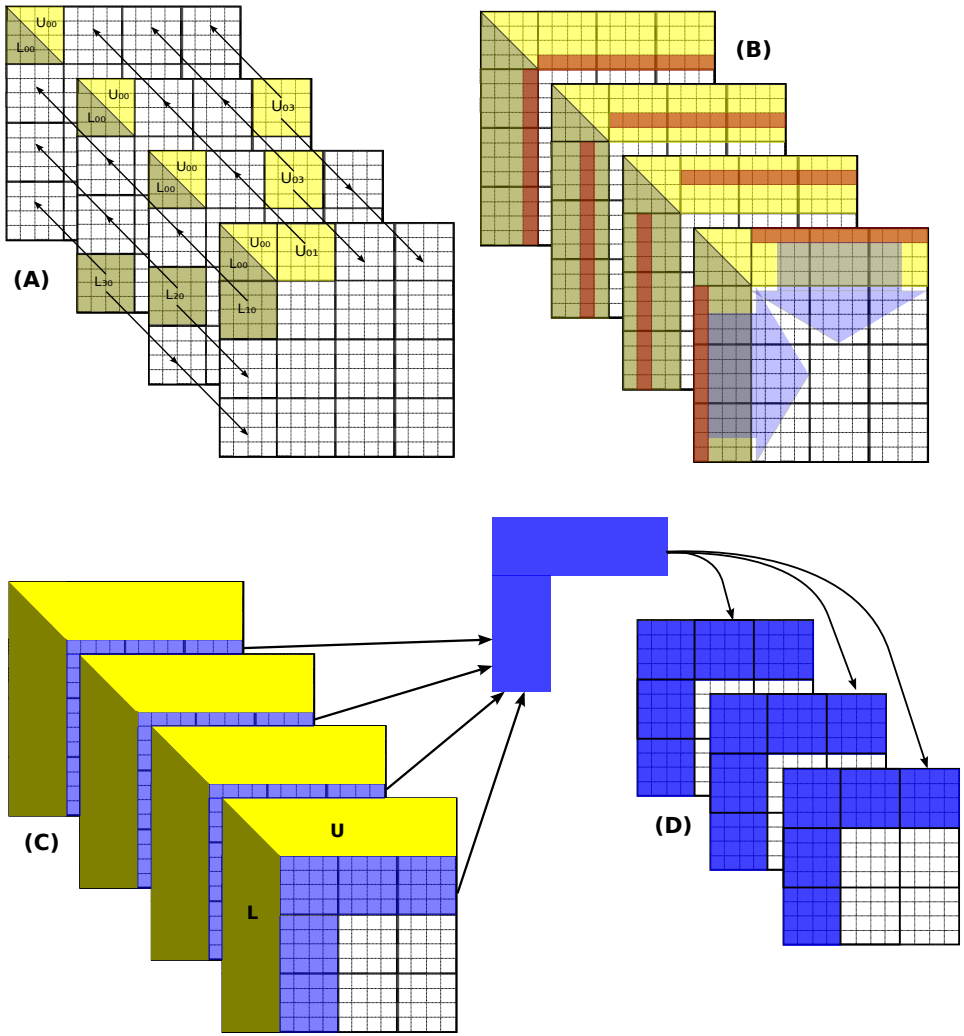
2.5D LU factorization



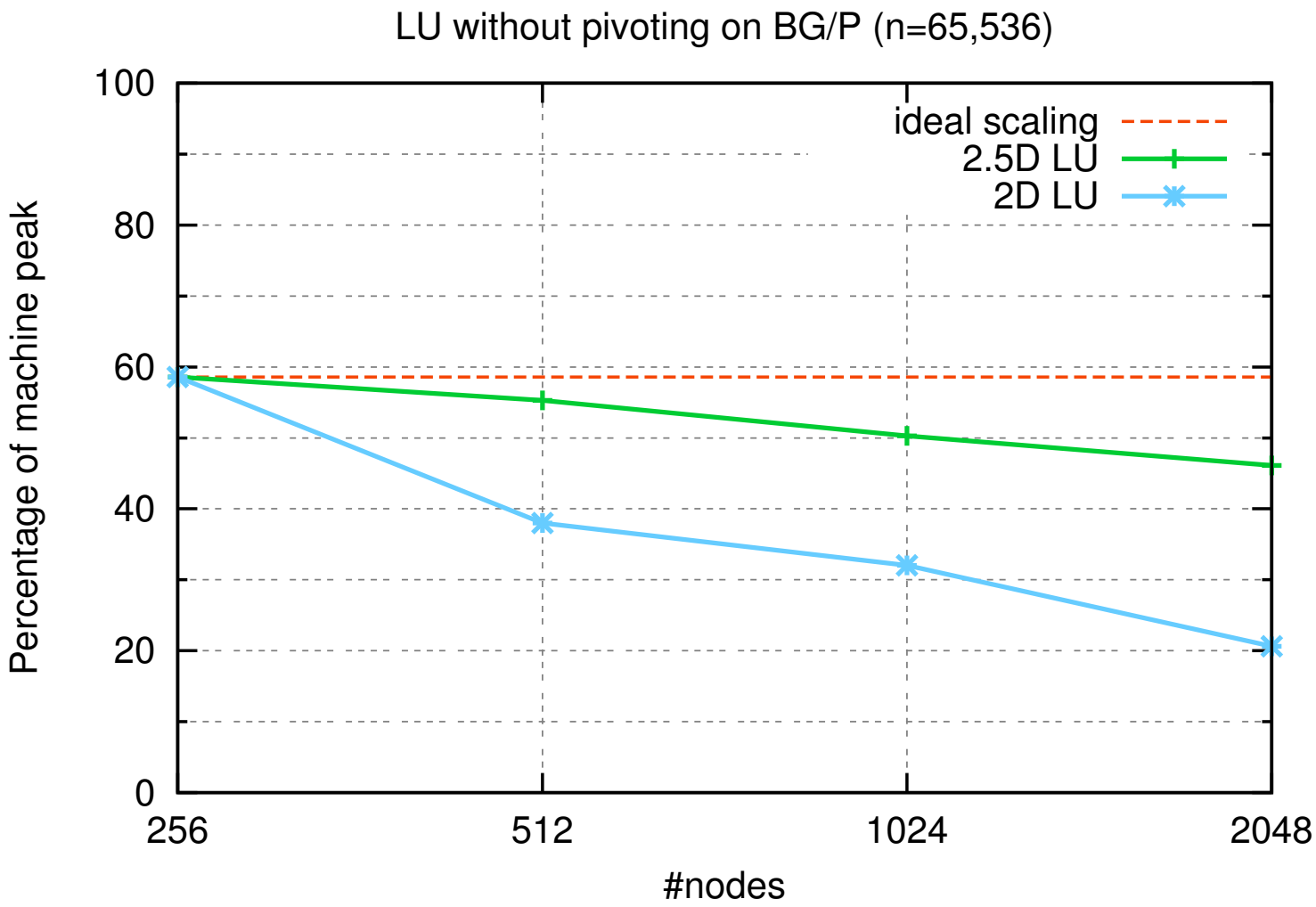
2.5D LU factorization



2.5D LU factorization



2.5D LU strong scaling (without pivoting)



2.5D LU with pivoting

$A = P \cdot L \cdot U$, where P is a permutation matrix

- 2.5D generic pairwise elimination (neighbor/pairwise pivoting or Givens rotations (QR)) [A. Tiskin 2007]
  - pairwise pivoting does not produce an explicit L
  - pairwise pivoting may have stability issues for large matrices
- Our approach uses tournament pivoting, which is more stable than pairwise pivoting and gives L explicitly
  - pass up rows of A instead of U to avoid error accumulation

Tournament pivoting

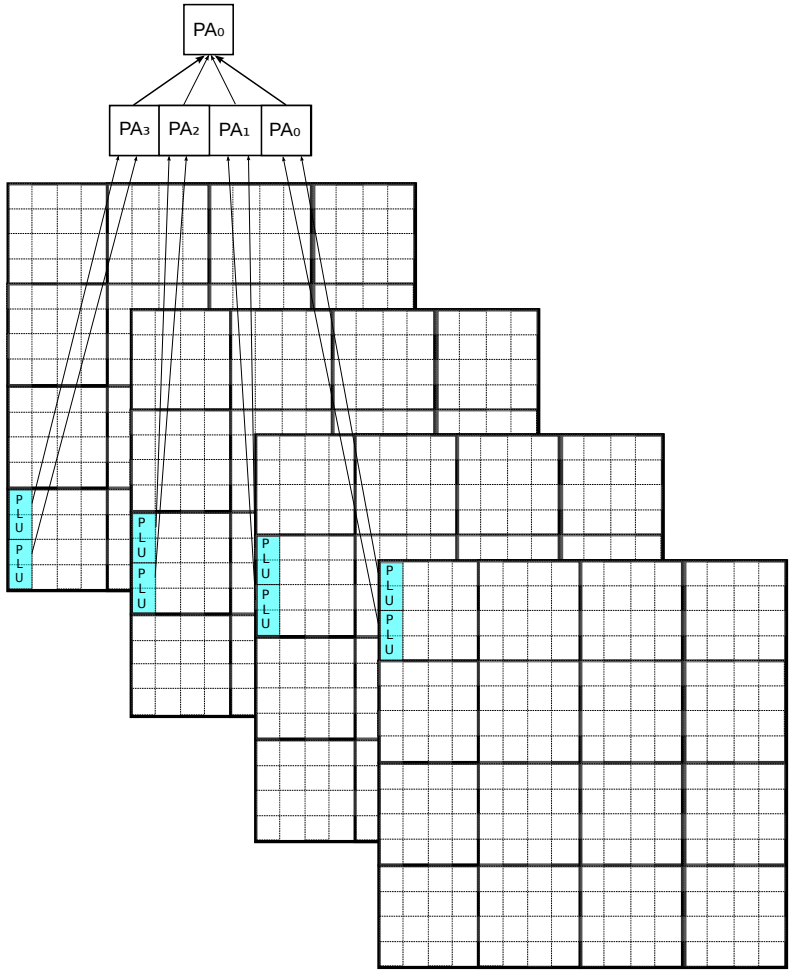
Partial pivoting is not communication-optimal on a blocked matrix

- requires message/synchronization for each column
- $O(n)$ messages needed

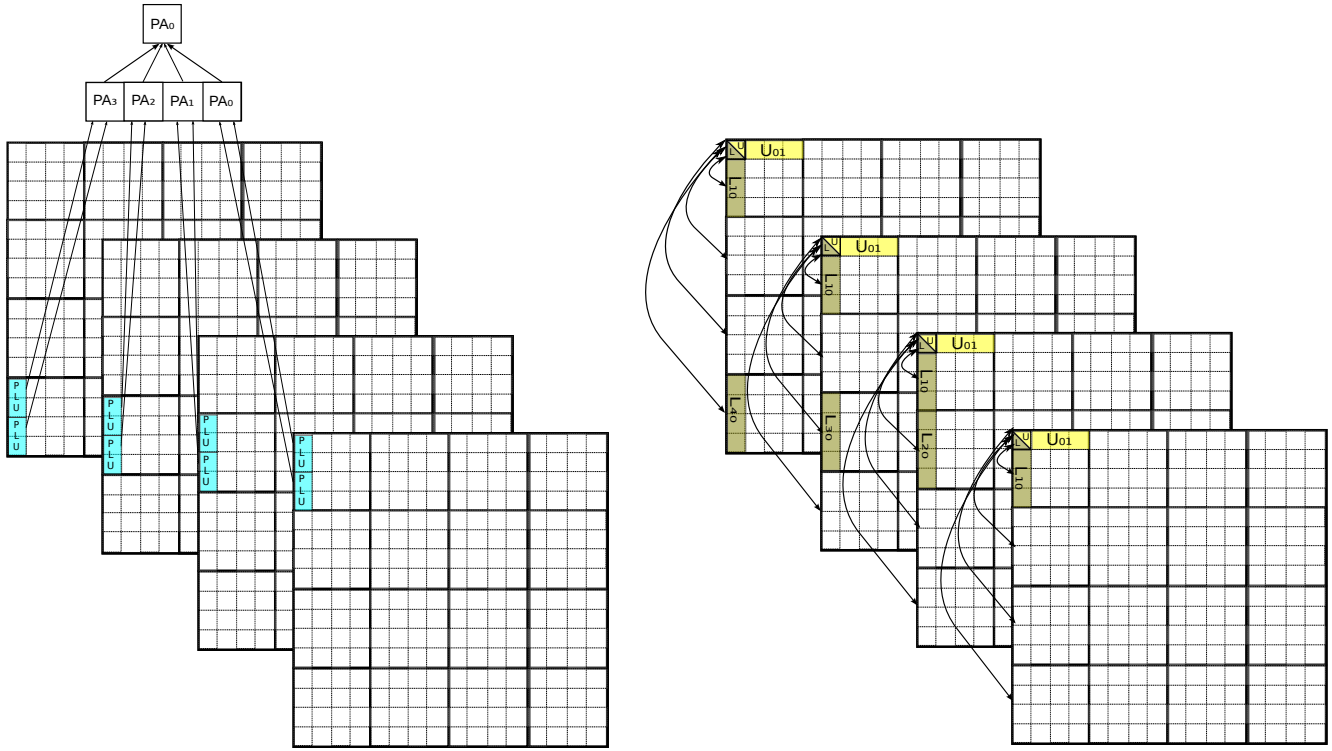
Tournament pivoting is communication-optimal

- performs a tournament to determine best pivot row candidates
- passes up 'best rows' of A

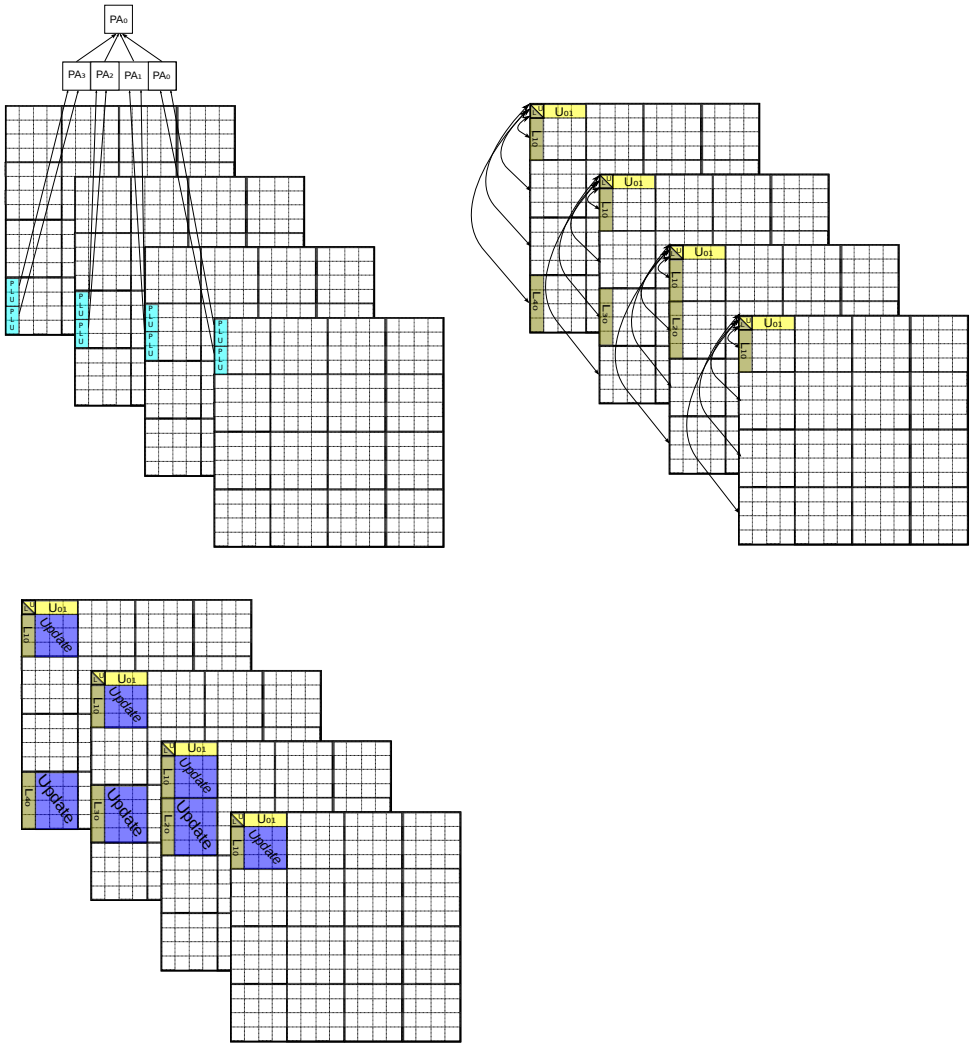
2.5D LU factorization with tournament pivoting



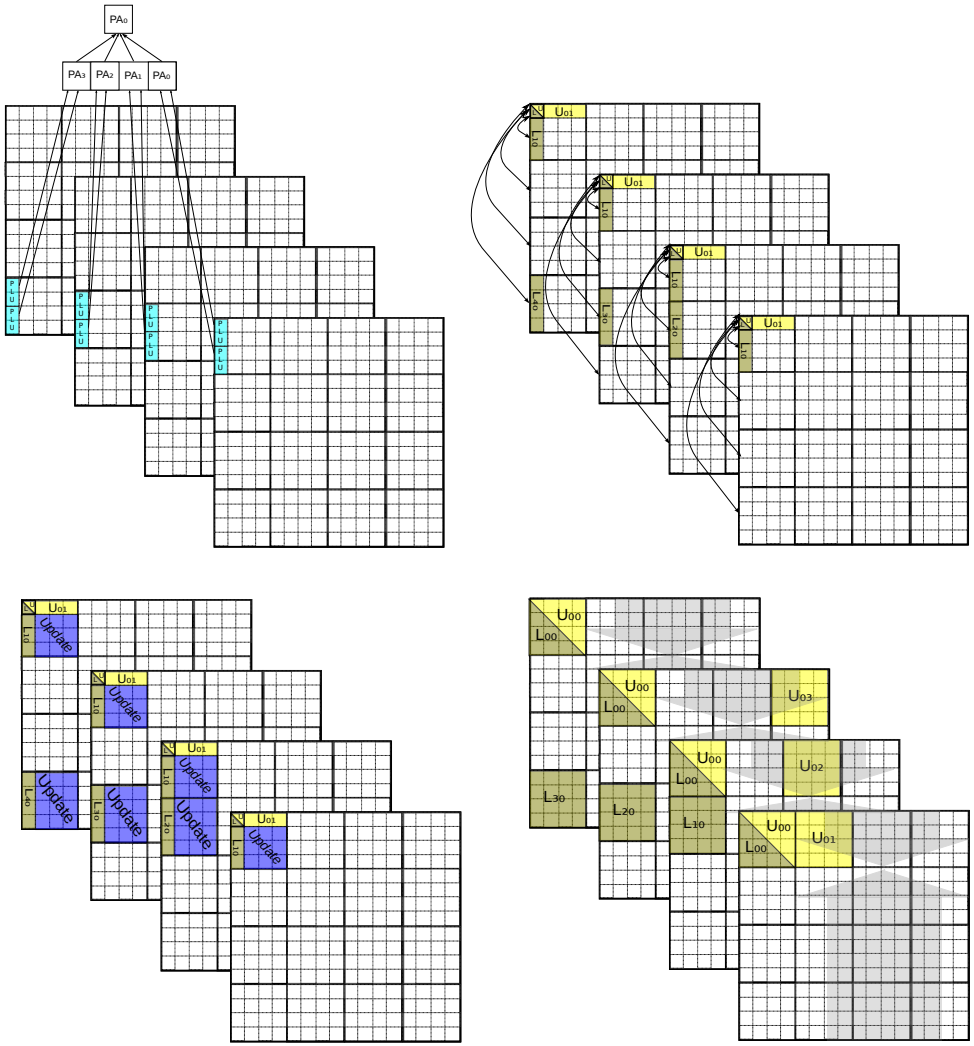
2.5D LU factorization with tournament pivoting



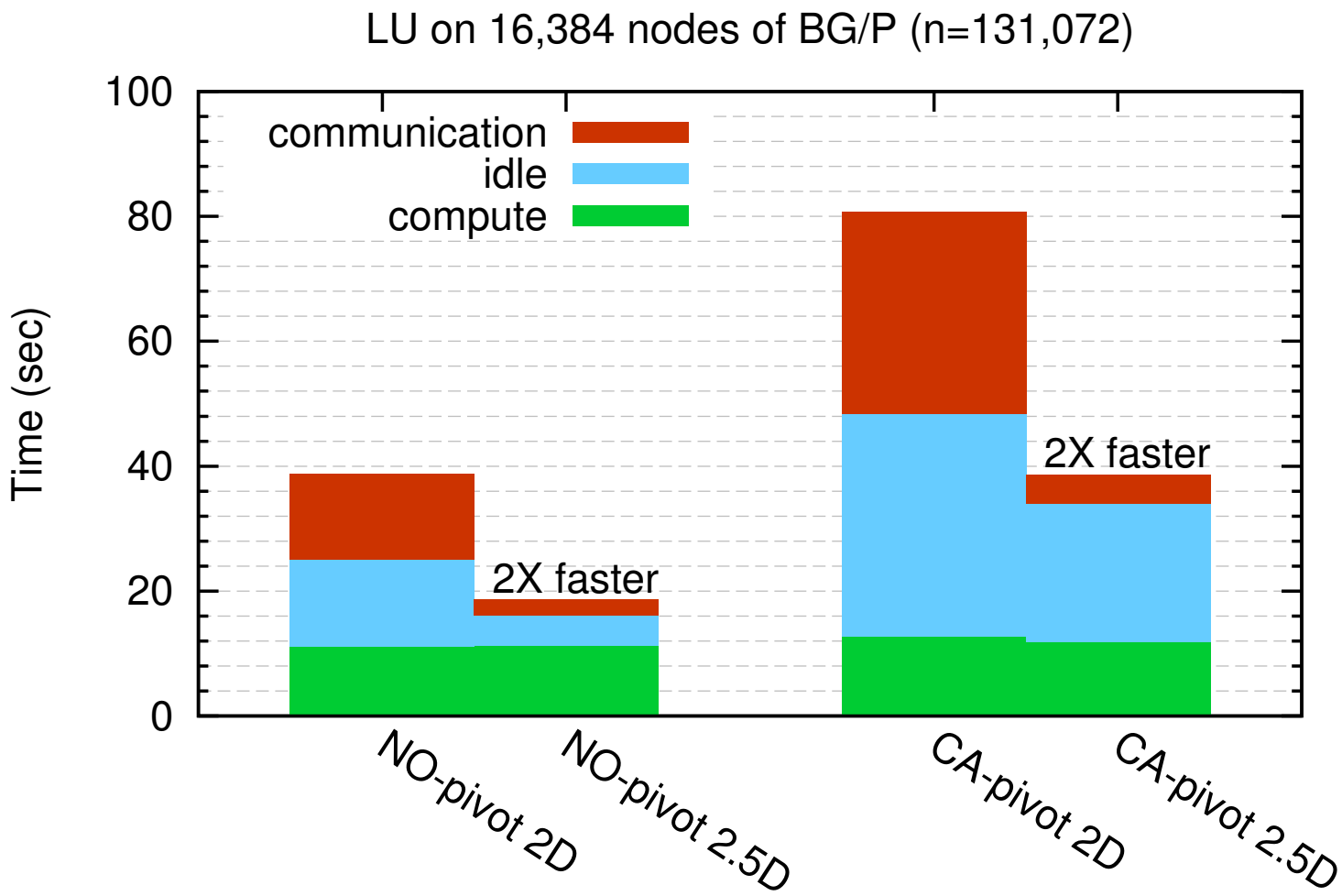
2.5D LU factorization with tournament pivoting



2.5D LU factorization with tournament pivoting



2.5D LU on 65,536 cores



Symmetric eigensolve via QR

To solve the symmetric eigenproblem on matrix A, we need to diagonalize

$$A = UDU^T$$

where U are the singular vectors and D is the singular values. This can be done by a series of two-sided orthogonal transformations

$$A = U_1 U_2 \ldots U_k D U_k^T \ldots U_2^T U_1^T$$

The process may be reduced to three stages: a QR factorization reducing to banded form, a reduction from banded to tridiagonal, and a tridiagonal eigensolve. We consider the QR, which is the most expensive step.

3D QR factorization

$A = Q \cdot R$ where Q is orthogonal R is upper-triangular

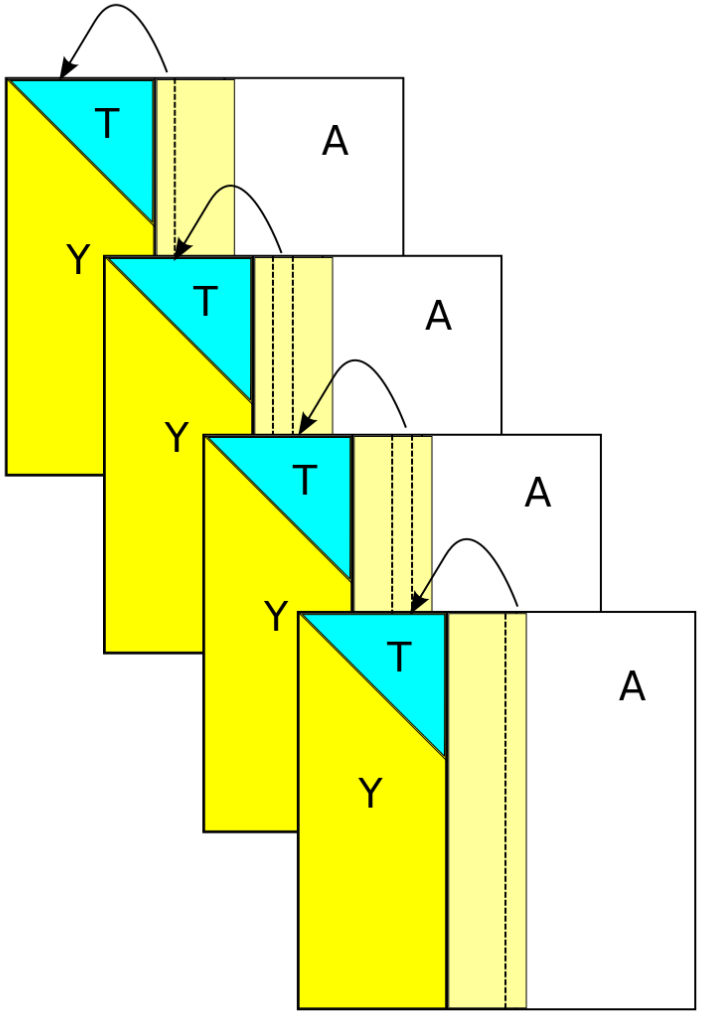
- 3D QR using Givens rotations (generic pairwise elimination) is given by [A. Tiskin 2007]
- Tiskin minimizes latency and bandwidth by working on slanted panels
- 3D QR cannot be done with right-looking updates as 2.5D LU due to non-commutativity of orthogonalization updates

3D QR factorization using the YT representation

The YT representation of Householder QR factorization is more work efficient when computing only R

- We give an algorithm that performs 2.5D QR using the YT representation
- The algorithm performs left-looking updates on Y
- Householder with YT needs fewer computation (roughly 2x) than Givens rotations

3D QR using YT representation



Latency-optimal 2.5D QR

To reduce latency, we can employ the TSQR algorithm

1. Given n-by-b panel partition into $2b$-by-b blocks
2. Perform QR on each $2b$-by-b block
3. Stack computed Rs into $n/2$-by-b panel and recursive
4. Q given in hierarchical representation

Need YT representation from hierarchical Q...

YT reconstruction

Yamamoto et al.

- Take Y to be the first b columns of Q minus the identity
- Define $T = (I - Q_1)^{-1}$
- Sacrifices triangular structure of T and Y.

Our first attempt

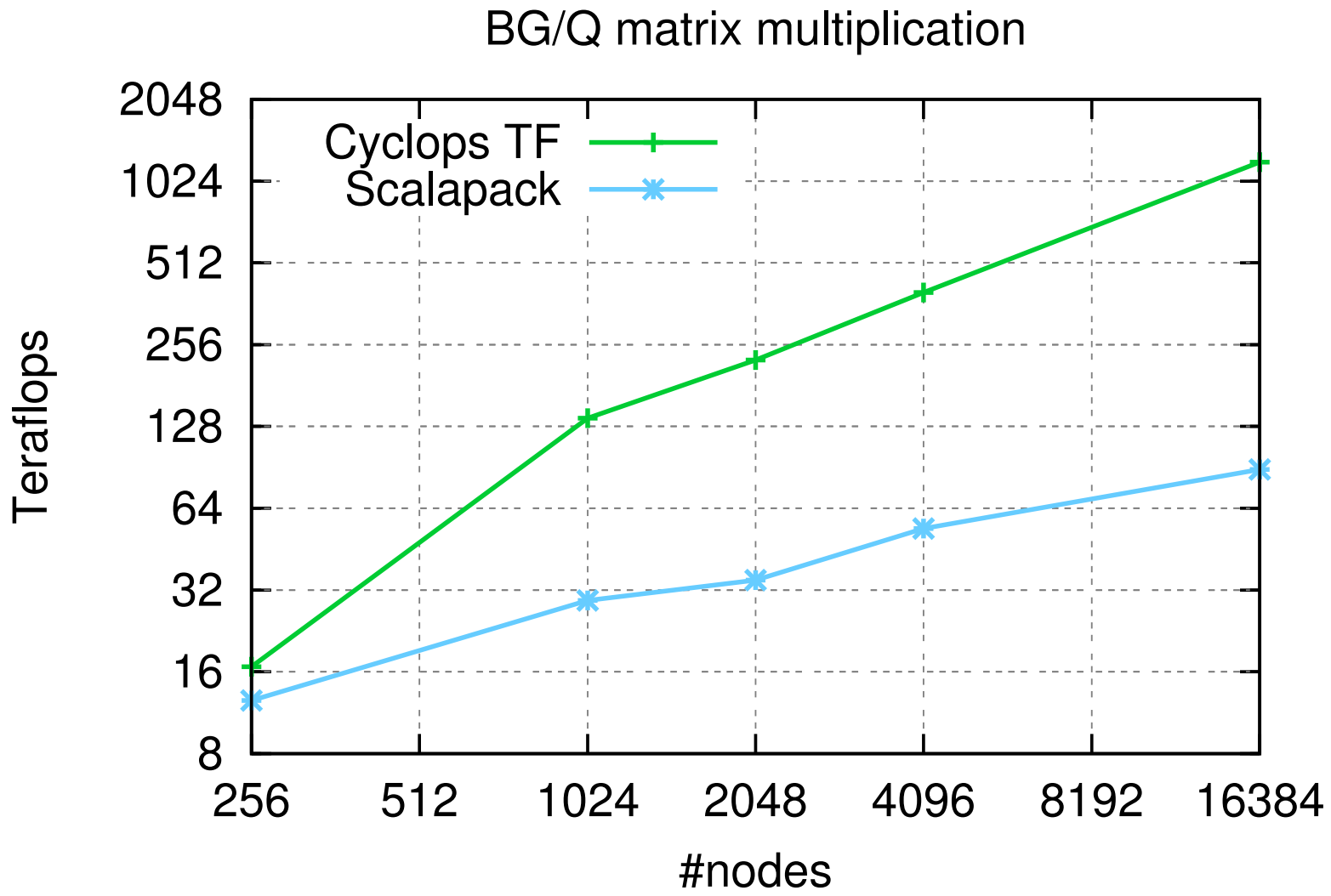
$$LU(R-A) = LU(R-(I-YTY^T)R) = LU(YTY^TR) = (Y)\cdot(TY^TR)$$

was unstable due to being dependent on the condition number of R. However, performing LU on Yamamoto's T seems to be stable,

$$LU(I-Q_1) = LU(I-(I-Y_1TY_1^T)) = LU(Y_1TY_1^T) = (Y_1)\cdot(TY_1^T)$$

and should yield triangular Y and T.

3D algorithms on BG/Q



3D algorithms for DFT

3D matrix multiplication is integrated into QBox.

- QBox is a DFT code developed by Erik Draeger et al.
- Depending on system/functional can spend as much as 80% time in MM
- Running on most of Sequoia and getting significant speed up from 3D
- 1.75X speed-up on 8192 nodes 1792 gold atoms, 31 electrons/atom
- Eventually hope to build and integrate a 3D eigensolver into QBox