DISTAL: Distributed Dense and Sparse Tensor Computations

Rohan Yadav, Alex Aiken, Fredrik Kjolstad

PETSc TAO

TRILINOS

ScaLAPACK

Predefined set of kernels

$$a = \mathbf{B} \cdot c$$

$$A = \mathbf{B} \cdot C \qquad \mathbf{A} = \mathbf{B} + \mathbf{C}$$

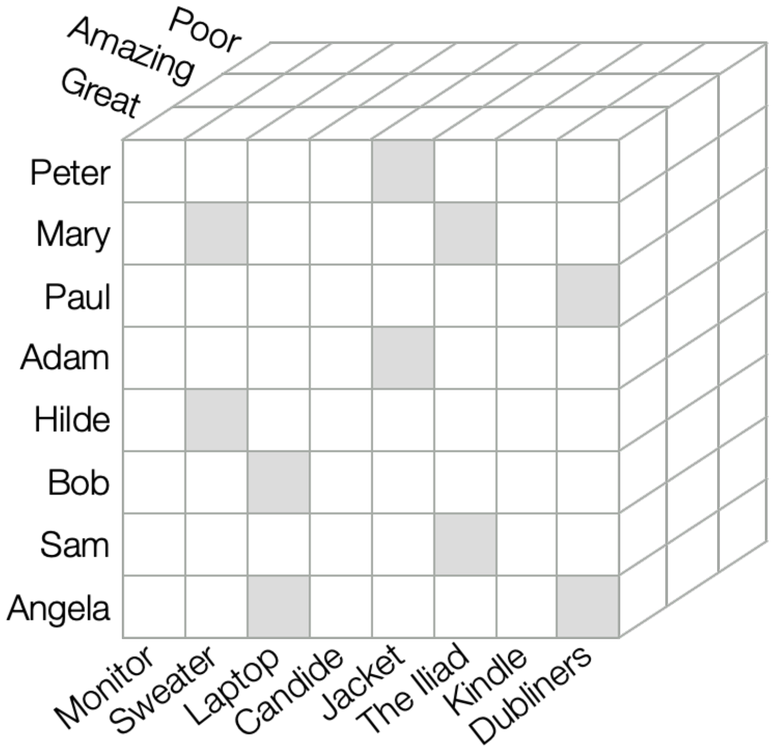
$$\ldots$$

Predefined set of formats/distributions

CSR / Tiled

Get close to peak performance!

Higher order data?



Kernels outside of predefined set?

$$\mathbf{A} = \mathbf{B} + \mathbf{C} + \mathbf{D}$$

$$\mathbf{A}_{ij} = \mathbf{B}_{ijk} \cdot c_k \qquad \mathbf{A}_{ij} = \mathbf{B}_{ij} \cdot C_{ik} \cdot D_{kj}$$

$$\ldots \qquad \ldots \qquad \ldots$$

Flexible data formats?

Cyclops Tensor Framework

parallel arithmetic on multidimensional arrays

All of tensor algebra

C["..."] += A["..."] * B["..."]

Leaves performance on the table

3-4x **slowdown** for dense
1-2 **orders of magnitude** for sparse

Programmers today face a trade-off!

Libraries of Kernels

ScaLAPACK

PETSc TAO

TRILINOS

Compilation

This work: DISTAL

Performance

Generality
(All of tensor algebra)

Factorization

Cyclops Tensor Framework
(CTF)

What to compute
(Tensor Index Notation)

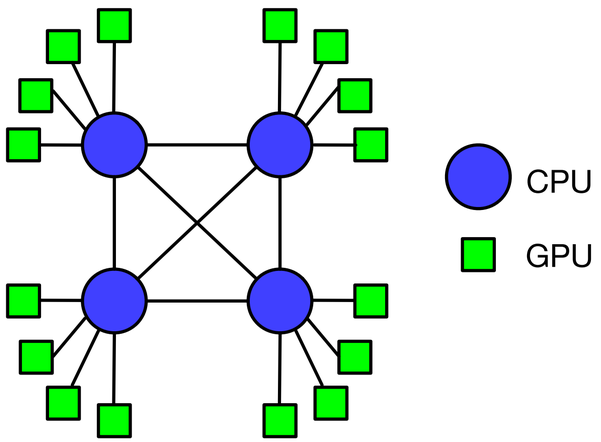
How data is compressed
(Format Language)

How computation is distributed
(Scheduling Language)

How data is distributed
(Tensor Distribution Notation)

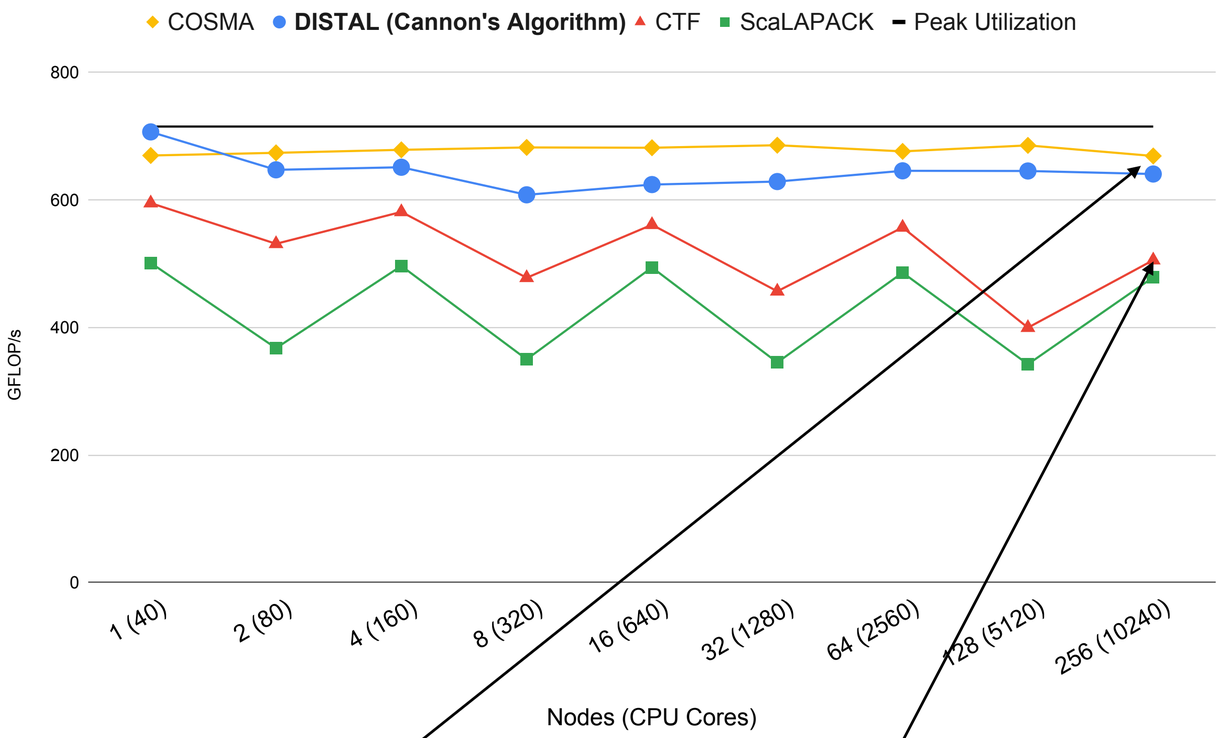
DISTAL

y(i) = A(i, j) * x(j)

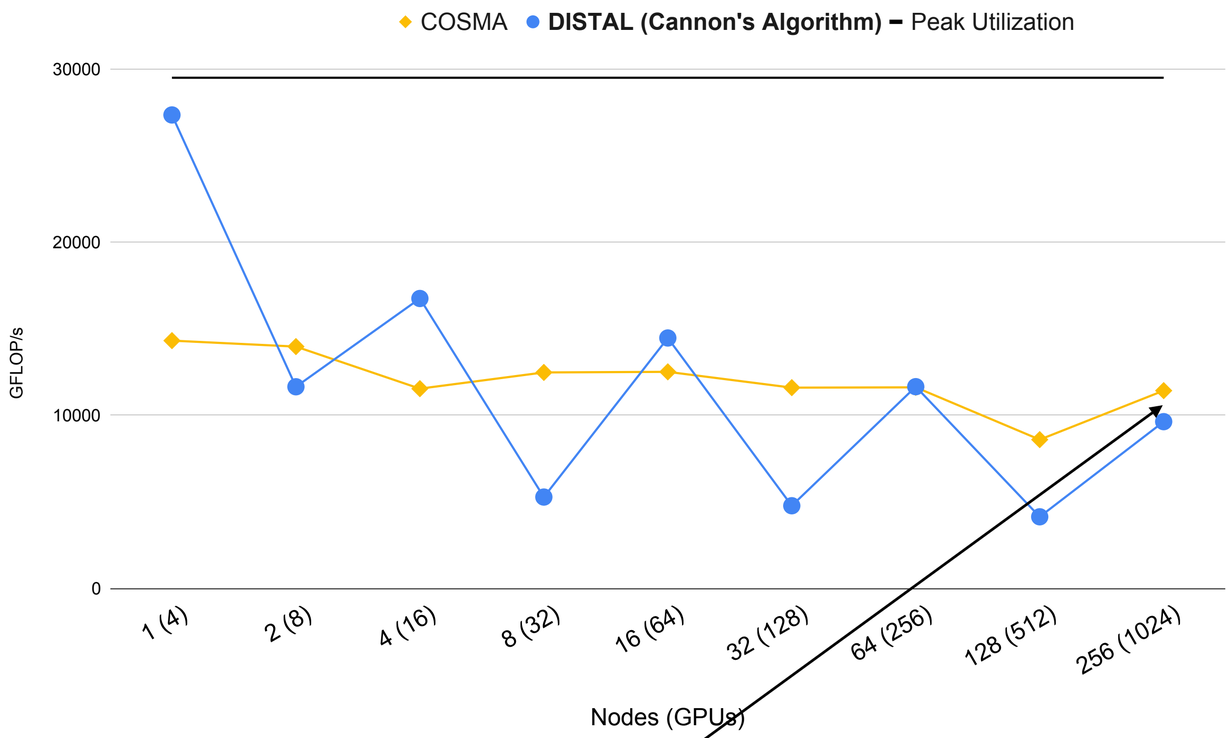


x,y distributed in chunks
A distributed row-wise

DISTAL Competes with Hand-Tuned Code

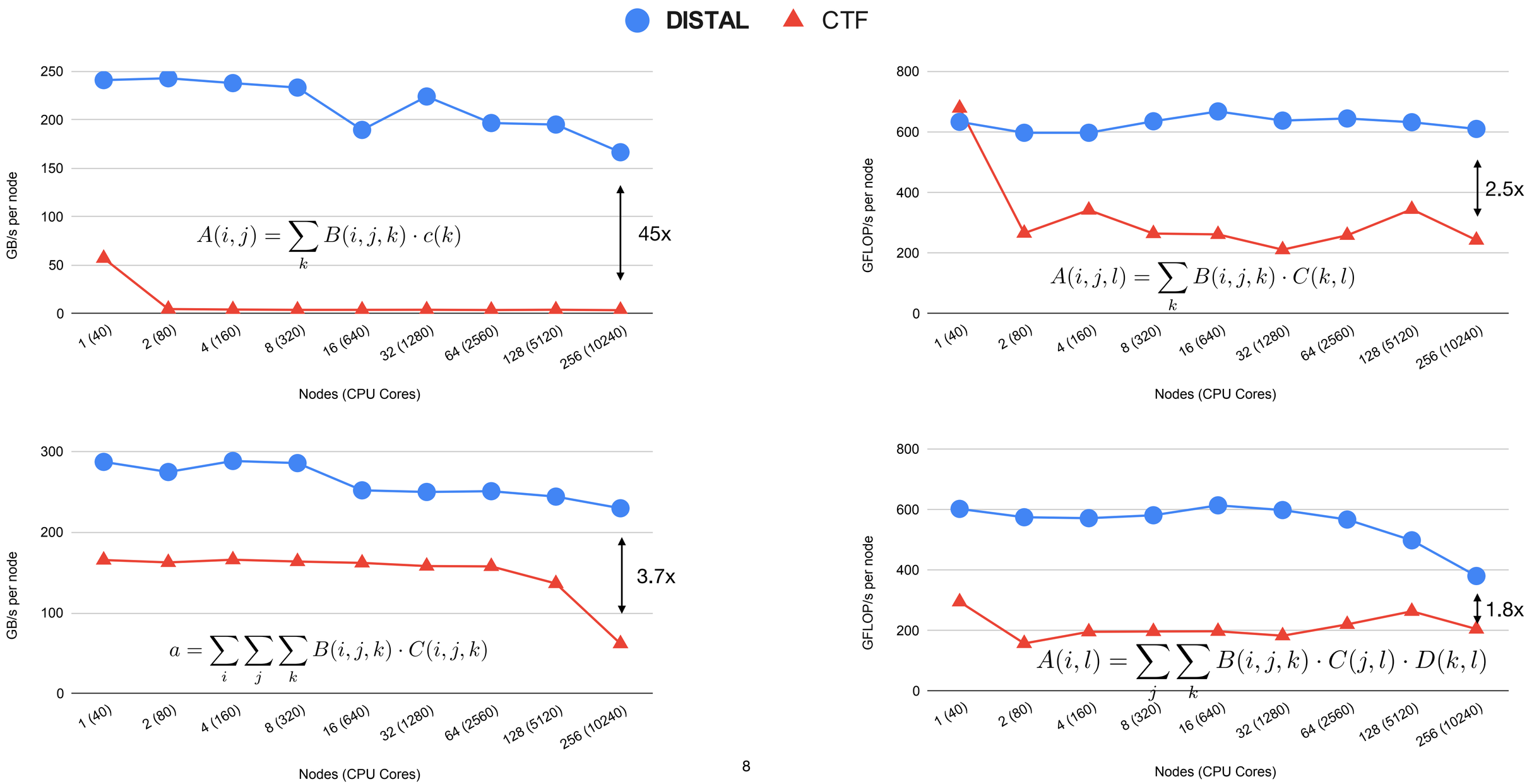


0.95x COSMA **1.25x** ScaLAPACK

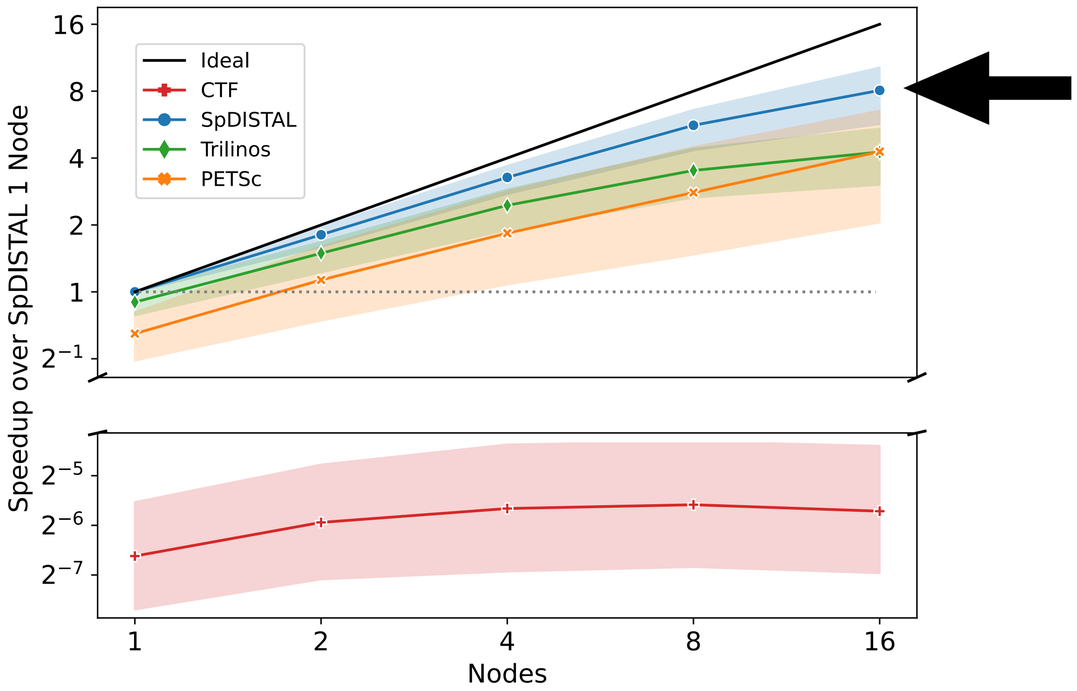


0.85x COSMA

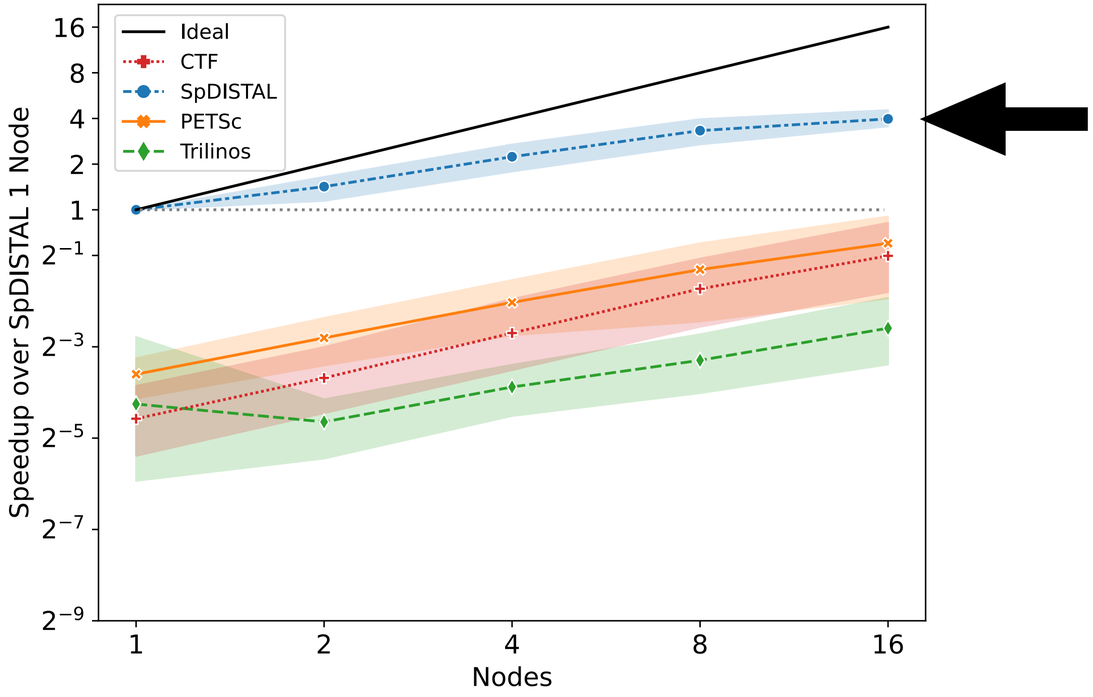
DISTAL Generalizes High Performance



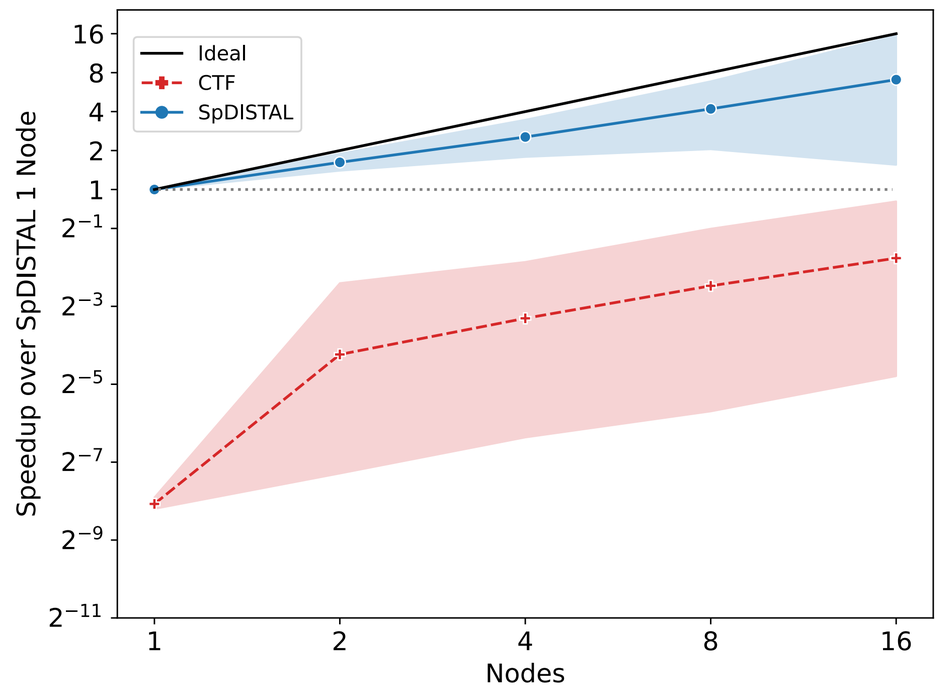
Same Trends for Sparse Tensors



$a = \mathbf{B} \cdot c$

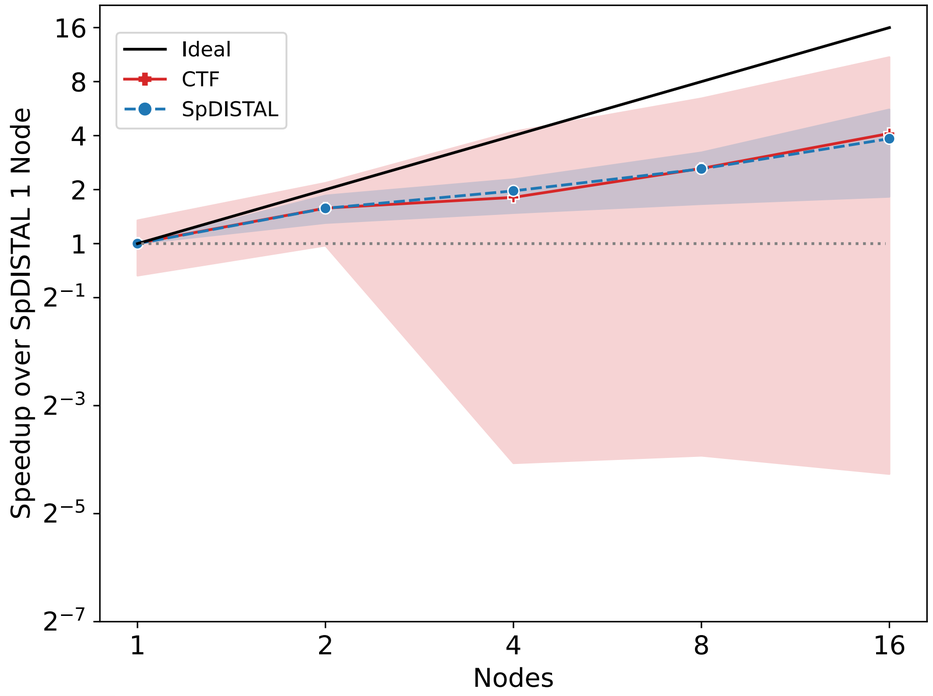


(Fused) $\mathbf{A} = \mathbf{B} + \mathbf{C} + \mathbf{D}$

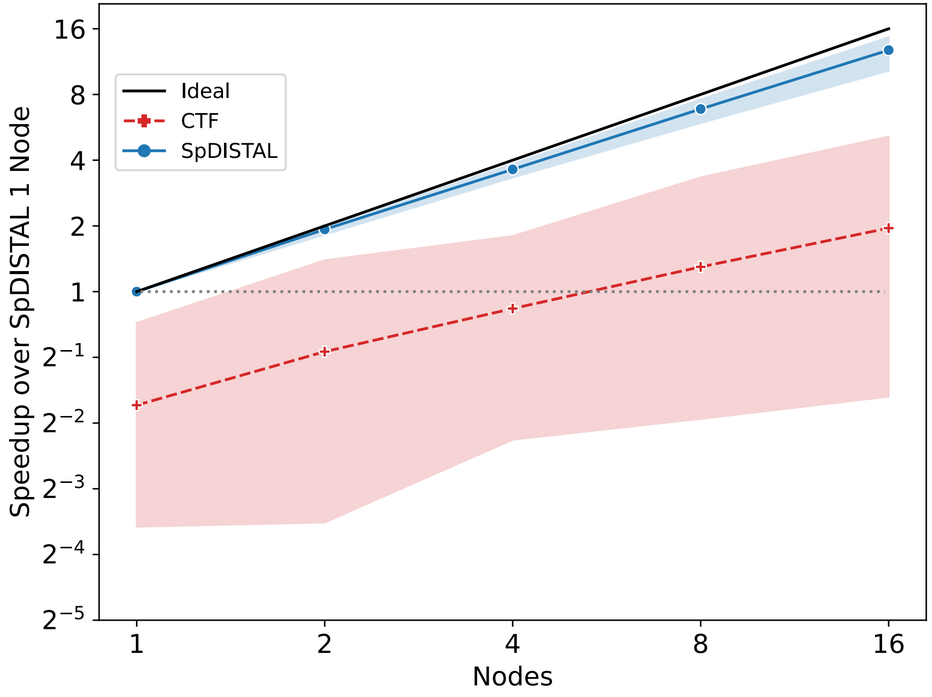


SpTTV

SDDMM



SpMTTKRP



Dense MatVec Example

```
Machine m(...);
int n = ...;

// Declare tensors. Data distribution elided.
Tensor<double> y({n}, {Dense}, ...);
Tensor<double> A({n, n}, {Dense, Dense}, ...);
Tensor<double> x({n}, {Dense}, ...);
```

Dense MatVec Example

```
Machine m(...);
int n = ...;

// Declare tensors. Data distribution elided.
Tensor<double> y({n}, {Dense}, ...);
Tensor<double> A({n, n}, {Dense, Dense}, ...);
Tensor<double> x({n}, {Dense}, ...);

// Declare computation.
IndexVar i, j;
y(i) = A(i, j) * x(j);
```

Dense MatVec Example

```
Machine m(...);
int n = ...;

// Declare tensors. Data distribution elided.
Tensor<double> y({n}, {Dense}, ...);
Tensor<double> A({n, n}, {Dense, Dense}, ...);
Tensor<double> x({n}, {Dense}, ...);

// Declare computation.
IndexVar i, j;
y(i) = A(i, j) * x(j);

// Schedule computation.
IndexVar io, ii;
y.schedule()
 .divide(i, io, ii, M.x)
 // Distribute outer loop iterations across all CPUs.
 .distribute(io, CPU)
 .communicate({y, A, x}, io)
 // Parallelize inner loop iterations across CPU threads.
 .parallelize(ii, CPUThread)
 ;
```

CSR Sparse MatVec Example

```
Machine m(...);
int n = ...;

// Declare tensors. Data distribution elided.
Tensor<double> y({n}, {Dense}, ...);
Tensor<double> A({n, n}, {Dense, Sparse}, ...);
Tensor<double> x({n}, {Dense}, ...);

// Declare computation.
IndexVar i, j;
y(i) = A(i, j) * x(j);

// Schedule computation.
IndexVar io, ii;
y.schedule()
 .divide(i, io, ii, M.x)
 // Distribute outer loop iterations across all CPUs.
 .distribute(io, CPU)
 .communicate({y, A, x}, io)
 // Parallelize inner loop iterations across CPU threads.
 .parallelize(ii, CPUThread)
 ;
```

Key Contributions of (Dense) DISTAL

What to compute

(Tensor Index Notation)

$$A(i,j) = \sum_k B(i,k) \cdot C(k,j)$$

How computation is distributed

(Scheduling Language)

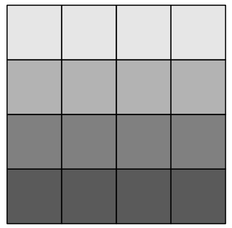
```
distribute(i)
communicate(B, i)
rotate(i, k)
```

How data is distributed

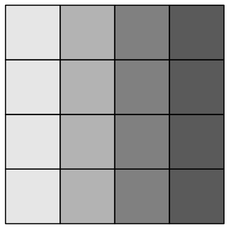
(Tensor Distribution Notation)

Describe partitioned tensor dimensions

$$\mathcal{T}_{\mathrm{xy} \mapsto \mathrm{x}}\, \mathcal{M}$$



$$\mathcal{T}_{\mathrm{xy} \mapsto \mathrm{y}}\, \mathcal{M}$$



DISTAL's Language is Expressive

Cannon's Algorithm (1969)

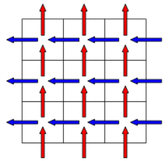
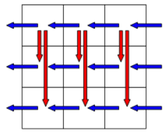
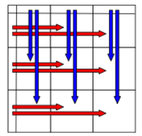
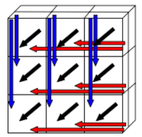
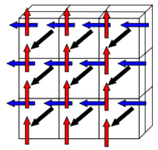
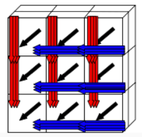
PUMMA (1994)

SUMMA (1995)

Johnson's Algorithm (1995)

Solomonik's Algorithm (2011)

COSMA (2019)

Comm. Pattern	Target Machine	Data Distribution	Schedule
	$\mathcal{M}(gx, gy)$	$A_{ij} \mapsto_{ij} \mathcal{M}$ $B_{ij} \mapsto_{ij} \mathcal{M}$ $C_{ij} \mapsto_{ij} \mathcal{M}$	.distribute({i, j}, {in, jn}, {il, jl}, Grid(gx, gy)) .divide(k, ko, ki, gx) .reorder({ko, il, jl, ki}) .rotate(ko, {in, jn}, kos) .communicate(A, jn) .communicate({B, C}, kos)
	$\mathcal{M}(gx, gy)$	$A_{ij} \mapsto_{ij} \mathcal{M}$ $B_{ij} \mapsto_{ij} \mathcal{M}$ $C_{ij} \mapsto_{ij} \mathcal{M}$	.distribute({i, j}, {in, jn}, {il, jl}, Grid(gx, gy)) .divide(k, ko, ki, gx) .reorder({ko, il, jl, ki}) .rotate(ko, {in}, kos) .communicate(A, jn) .communicate({B, C}, kos)
	$\mathcal{M}(gx, gy)$	$A_{ij} \mapsto_{ij} \mathcal{M}$ $B_{ij} \mapsto_{ij} \mathcal{M}$ $C_{ij} \mapsto_{ij} \mathcal{M}$	.distribute({i, j}, {in, jn}, {il, jl}, Grid(gx, gy)) .split(k, ko, ki, chunkSize) .reorder({ko, il, jl, ki}) .communicate(A, jn) .communicate({B, C}, ko)
	$\mathcal{M}(\sqrt[3]{p}, \sqrt[3]{p}, \sqrt[3]{p})$	$A_{ij} \mapsto_{ij0} \mathcal{M}$ $B_{ik} \mapsto_{i0k} \mathcal{M}$ $C_{kj} \mapsto_{0jk} \mathcal{M}$	.distribute({i, j, k}, {in, jn, kn}, {il, jl, kl}, Grid($\sqrt[3]{p}$, $\sqrt[3]{p}$, $\sqrt[3]{p}$)) .communicate({A, B, C}, kn)
	$\mathcal{M}(\sqrt{\frac{p}{c}}, \sqrt{\frac{p}{c}}, c)$	$A_{ij} \mapsto_{ij0} \mathcal{M}$ $B_{ij} \mapsto_{ij0} \mathcal{M}$ $C_{ij} \mapsto_{ij0} \mathcal{M}$	.distribute({i, j, k}, {in, jn, kn}, {il, jl, kl}, Grid($\sqrt{\frac{p}{c}}$, $\sqrt{\frac{p}{c}}$, c)) .divide(kl, k1, k2, $\sqrt{\frac{p}{c^3}}$) .reorder({k1, il, jl, k2}) .rotate(k1, {in, jn}, k1s) .communicate(A, jn) .communicate({B, C}, k1s)
	induced by schedule	induced by schedule	// gx, gy, gz, numSteps computed by COSMA scheduler. .distribute({i, j, k}, {in, jn, kn} {il, jl, kl}, Grid(gx, gy, gz)) .divide(kl, klo, kli, numSteps) .reorder({klo, il, jl, kli}) .communicate(A, kn) .communicate({B, C}, klo)

```
for i:
  for j:
    y(i) = A(i, j) * x(j)
```

```
for io:
  for ii:
    for j:
      i = io * ... + ii
      y(i) = A(i, j) * x(j)
```

```
y.schedule()
  .divide(i, io, ii, M.x)
  // Distribute outer loop iterations across all CPUs.
  .distribute(io, CPU)
  .communicate({y, A, x}, io)
  // Parallelize inner loop iterations across CPU threads.
  .parallelize(ii, CPUThread)
  ;
```

```
index launch io:
  for ii:
    for j:
      i = io * ... + ii
      y(i) = A(i, j) * x(j)
```

```
y.schedule()
 .divide(i, io, ii, M.x)
 // Distribute outer loop iterations across all CPUs.
 .distribute(io, CPU)
 .communicate({y, A, x}, io)
 // Parallelize inner loop iterations across CPU threads.
 .parallelize(ii, CPUThread)
 ;
```

```
y_part = partition(y)
A_part = partition(A)
index launch io over (y_part, A_part):
  y = y_part[io]
  A = A_part[io]
  for ii:
    for j:
      i = io * ... + ii
      y(i) = A(i, j) * x(j)
```

```
y.schedule()
 .divide(i, io, ii, M.x)
 // Distribute outer loop iterations across all CPUs.
 .distribute(io, CPU)
 .communicate({y, A, x}, io)
 // Parallelize inner loop iterations across CPU threads.
 .parallelize(ii, CPUThread)
 ;
```

What does “partition” do when A is sparse?

```
y_part = partition(y)
A_part = partition(A)
index launch io over (y_part, A_part):
  y = y_part[io]
  A = A_part[io]
  for ii:
    for j:
      i = io * ... + ii
      y(i) = A(i, j) * x(j)
```

Generation of data partitioning code is the key challenge

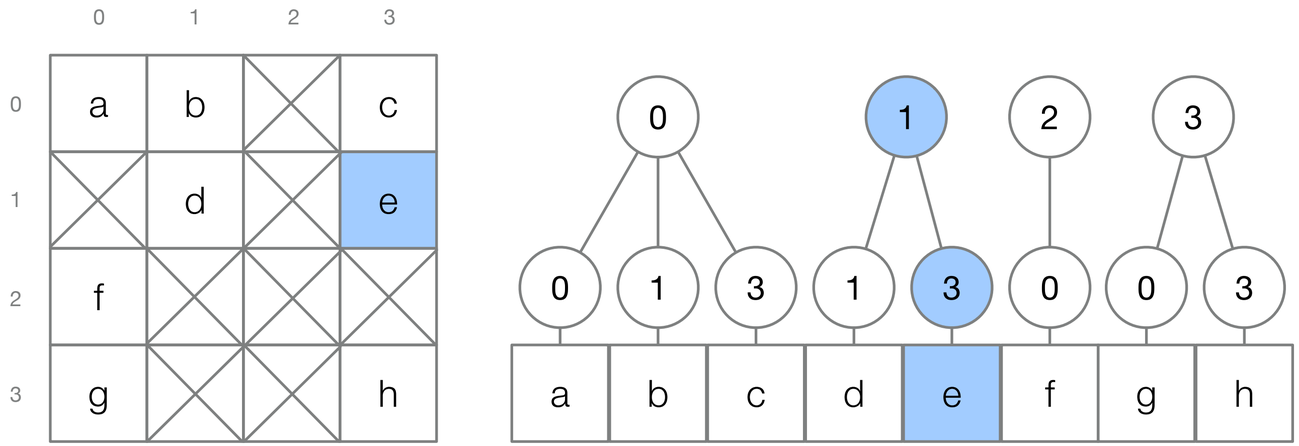
Decompose partitioning into a per-dimension problem

Use hybrid static and dynamic analyses

Static: generate code that describes the structure of data partitions

Dynamic: at runtime, use dynamic analysis to compute precise partitions

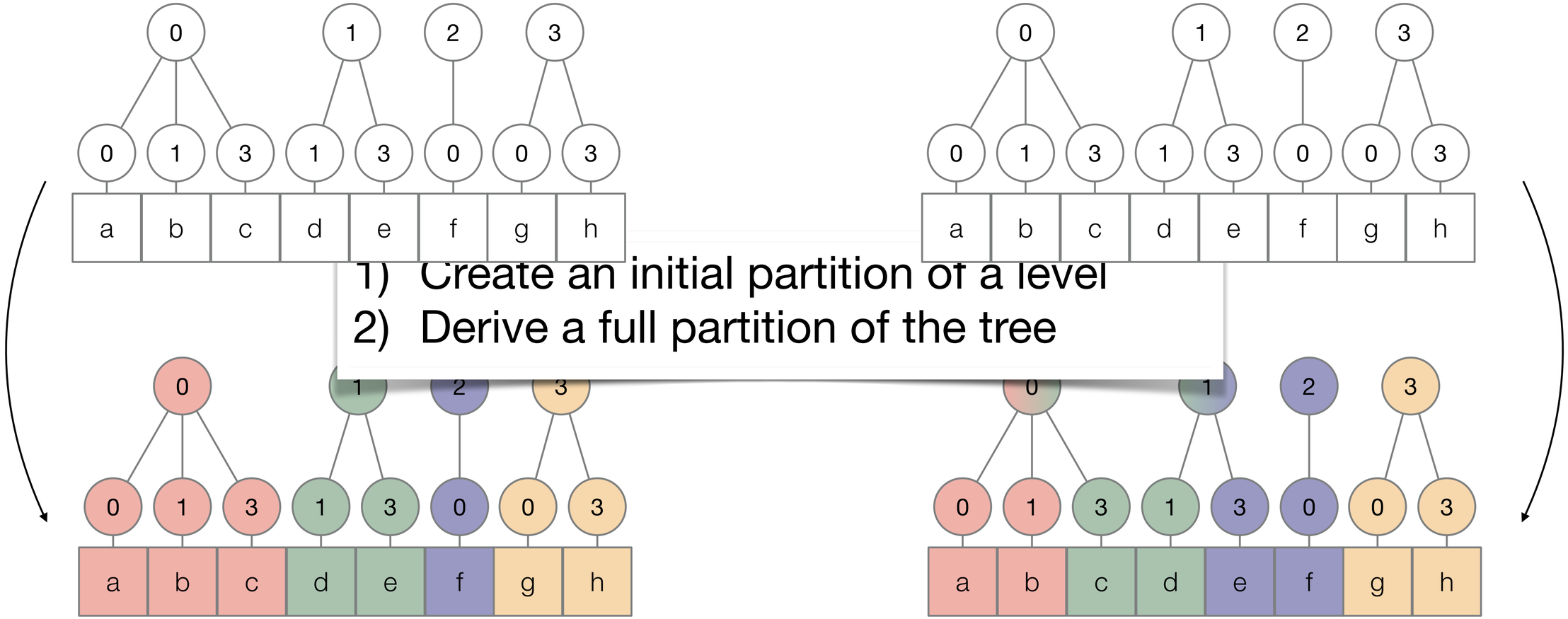
Intuition: Coordinate Trees



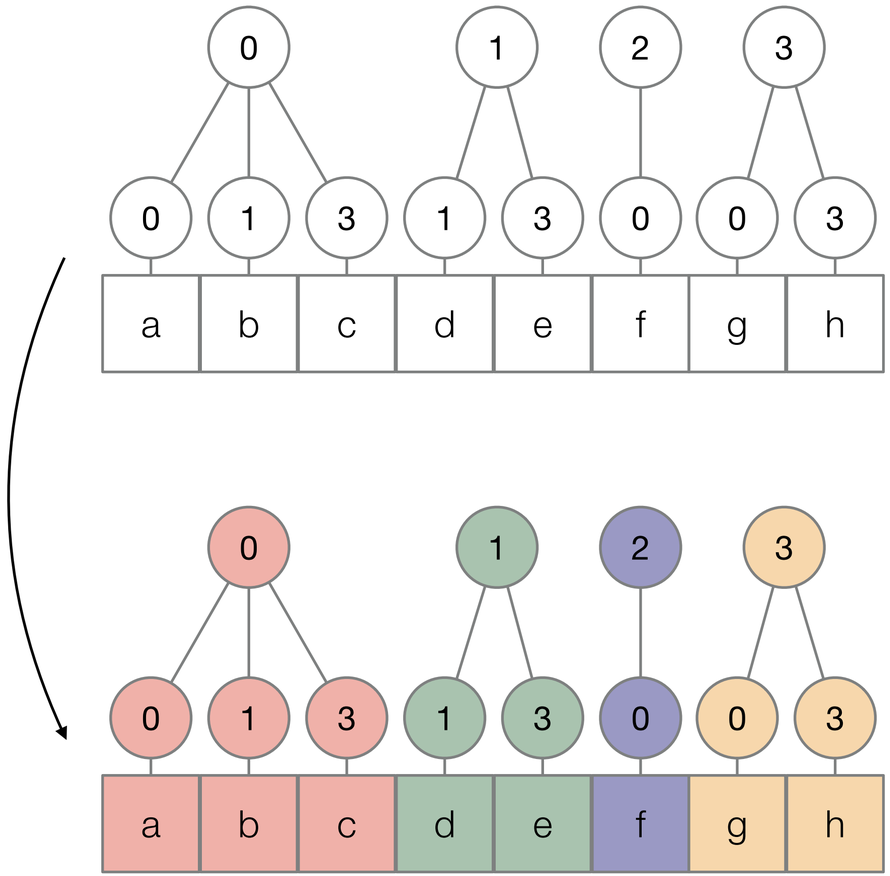
{Dense, Sparse}

size	4							
pos	0-2	3-4	5-5	6-7				
crd	0	1	3	1	3	0	0	3
vals	a	b	c	d	e	f	g	h

Partitioning Coordinate Trees



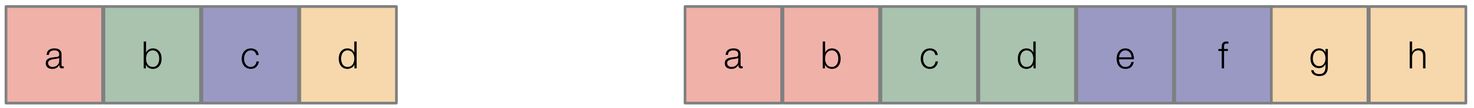
1) Create an initial partition of a level
2) Derive a full partition of the tree



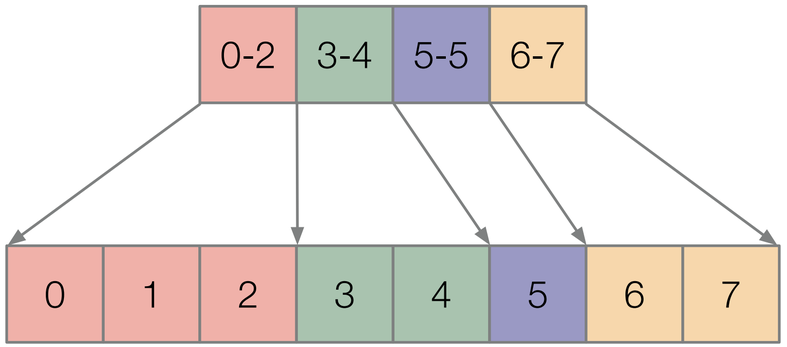
size	4							
pos	0-2	3-4	5-5	6-7				
crd	0	1	3	1	3	0	0	3
vals	a	b	c	d	e	f	g	h

Partitioning Operators

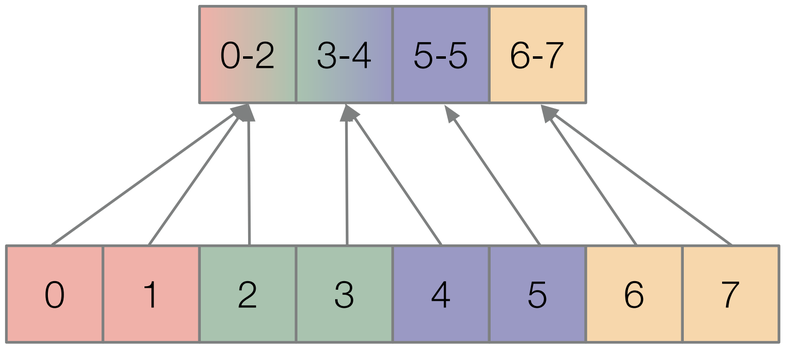
tiling(R, tile_size)



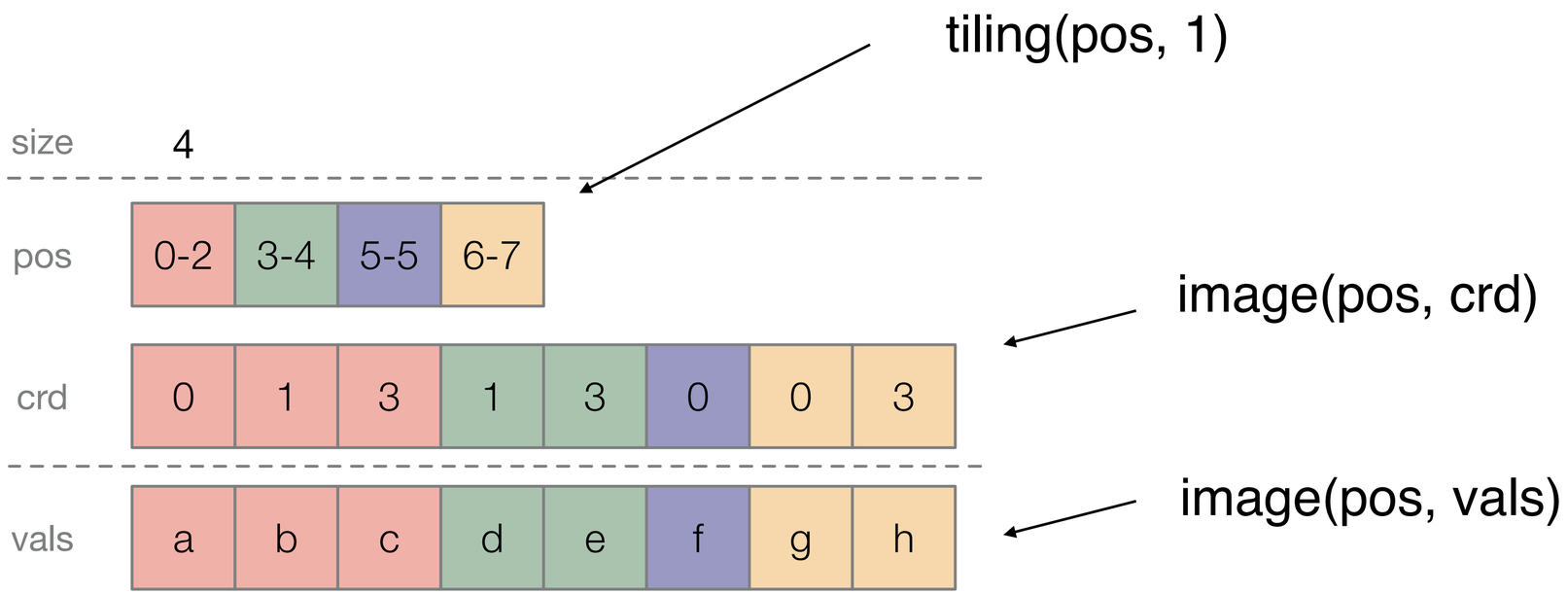
image(R1, R2)



preimage(R1, R2)



Partitioning Sparse Tensors



How can we generate these partitioning calls?

Data Format Describes Partitioning

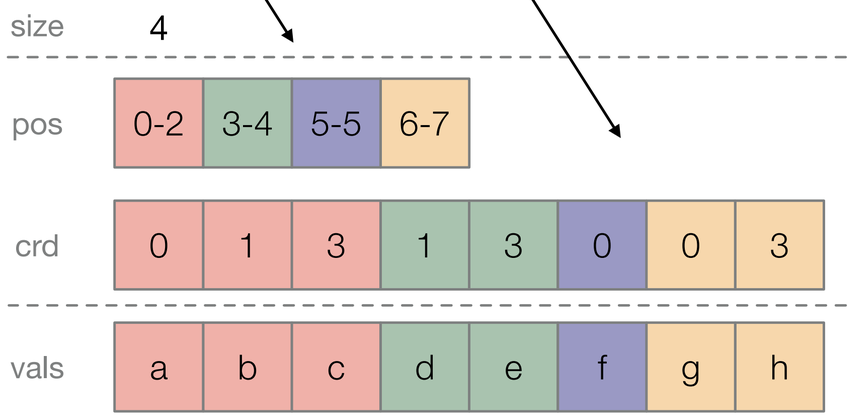
Formats implement a compile time interface describing partitioning capabilities

```
type Format {
  ir::Stmt initial_partition(…)
  ir::Stmt partition_from_parent(…)
  ir::Stmt partition_from_child(…)
  … <and more> …
}
```

```
void RowSplitSpMV(y, A, x) {
  …
  A_pos_part = tiling(A.pos, …)
  A_crd_part = image(A_pos_part, A.crd)
  A_val_part = image(A_pos_part, A.val)
  …
  index launch (
    A_pos_part, A_crd_part, A_val_part
  ) {
    < … Optimized SpMV Kernel … >
  }
}
```

A.levels[1].initial_partition()

A.levels[2].partition_from_parent()



y(i) = A(i, j) * x(j)

distribute(i)

x = y = {Dense}
A = {Dense, Sparse}

DISTAL compiles dense and sparse tensor computations onto modern distributed machines

DISTAL can express many algorithms, generalize to tensor algebra

DISTAL utilizes Legion's partitioning capabilities to provide simple compilation interfaces

Contact: rohany@cs.stanford.edu

rohany.github.io