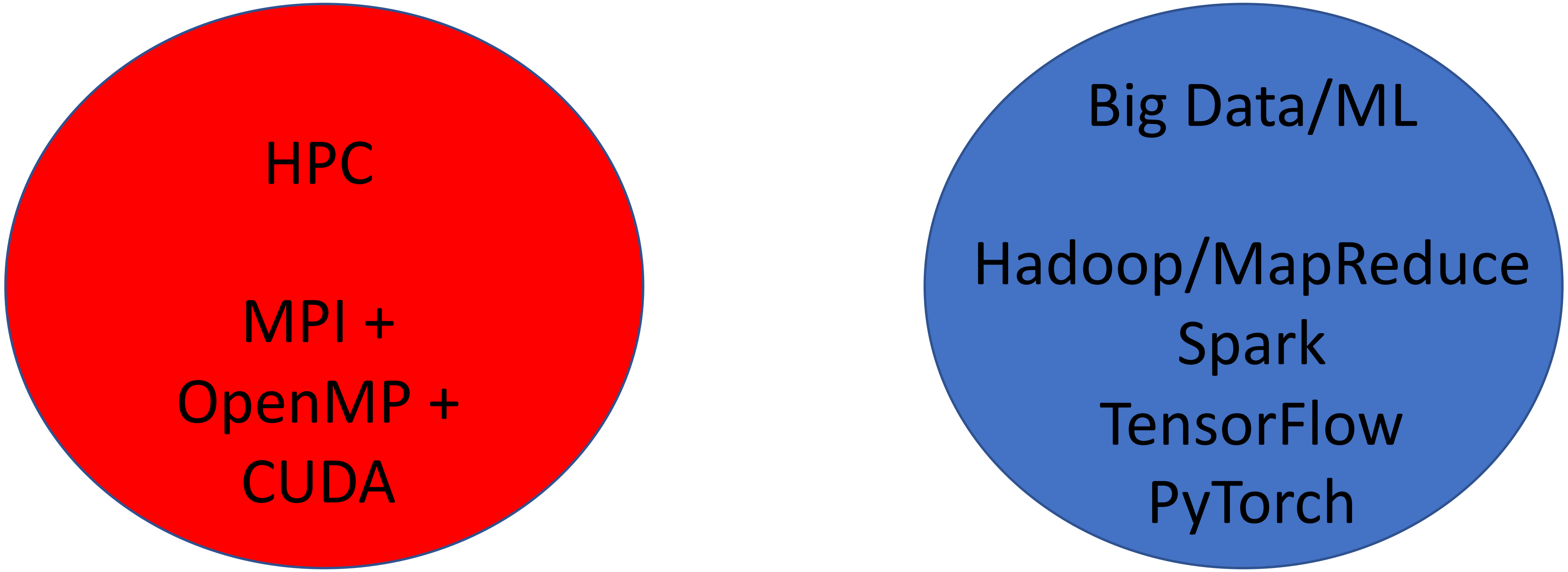
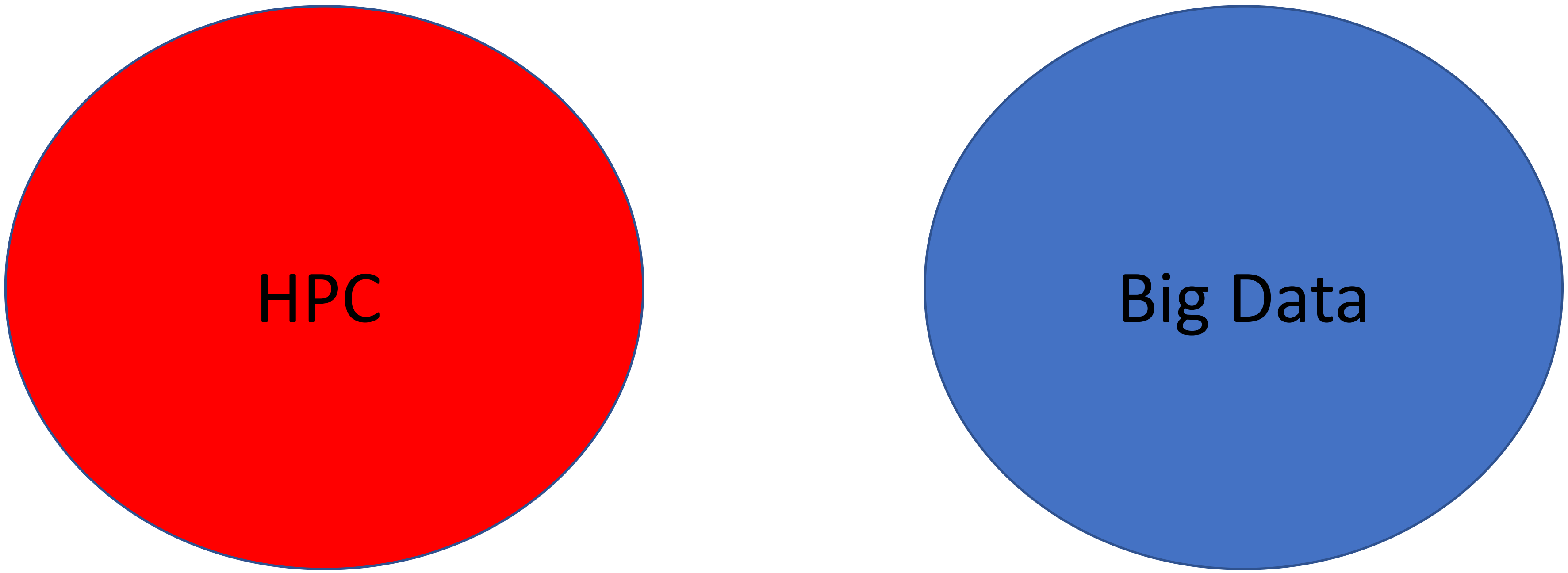
A Tale of Two Cultures

Alex Aiken
Stanford/SLAC

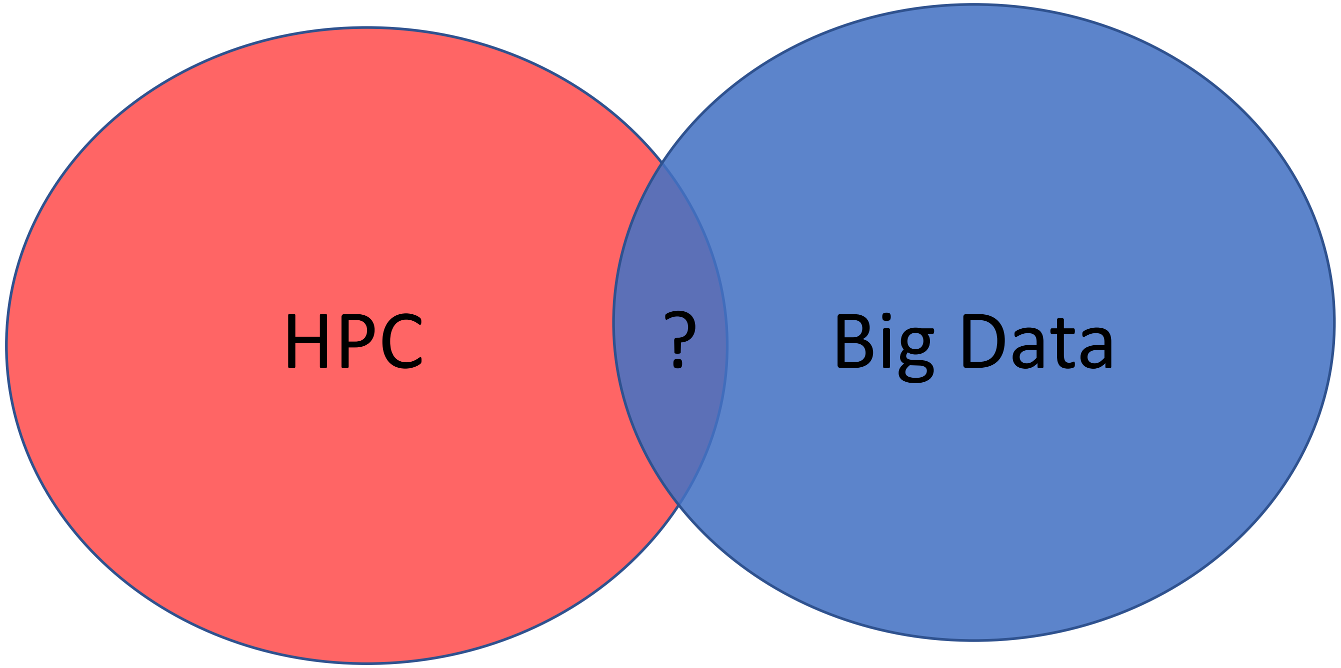
A Tale of Two Software Cultures



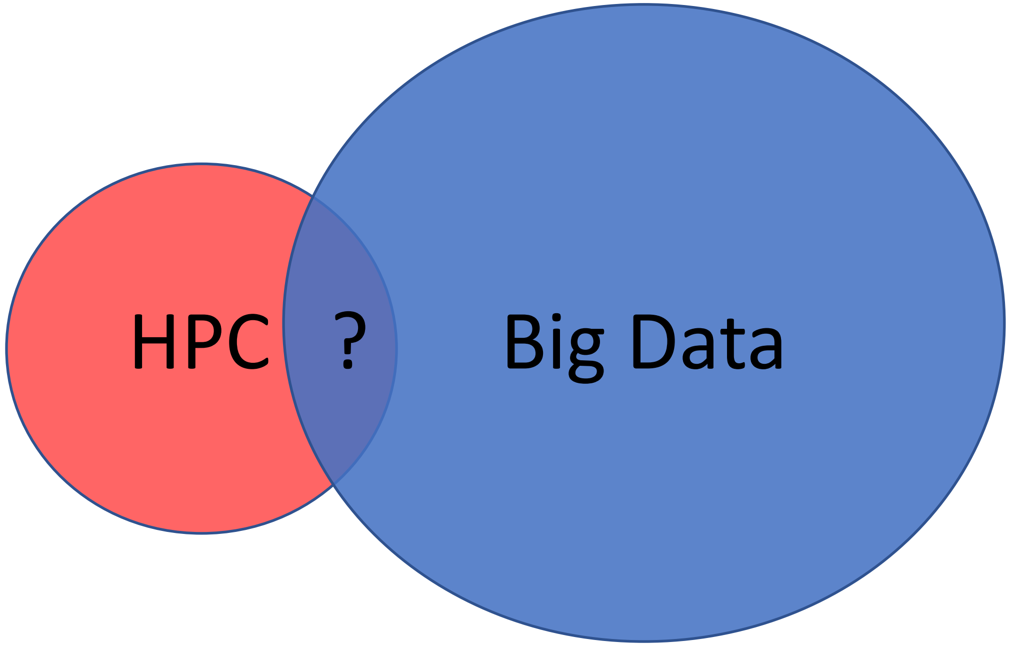
Is There Any Relationship?



Some Overlap …



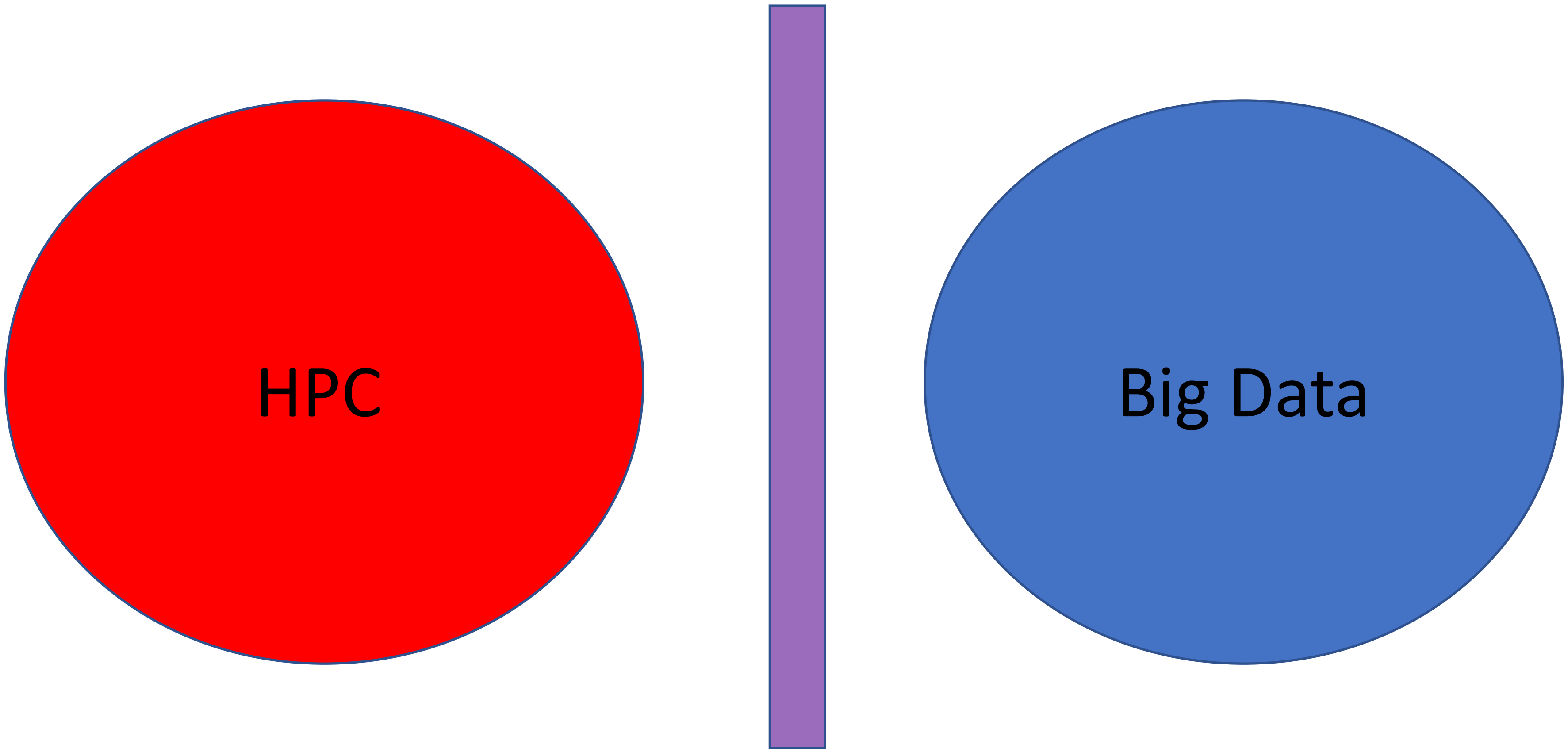
Some Difference in Size …



Will These Communities Converge?

- The stage is set: The underlying hardware is (almost) the same
- More shortly …

Are There Barriers to Convergence?



Priorities

HPC

- Performance
- Productivity
- Correctness

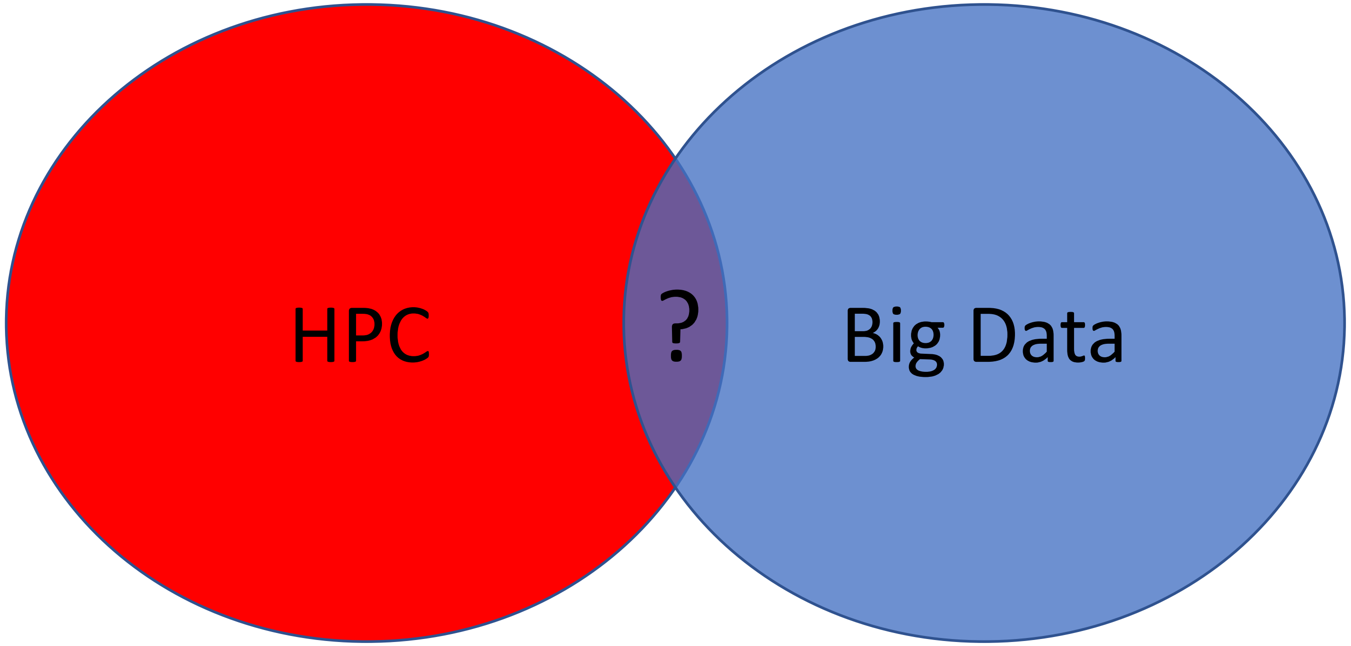
Big Data

- Productivity
- Performance
- Correctness

Creates Significant Differences In …

- Platform performance & programmer productivity
  - Obviously!
- Scale of computations
- Economic model

Is There Overlap Today?



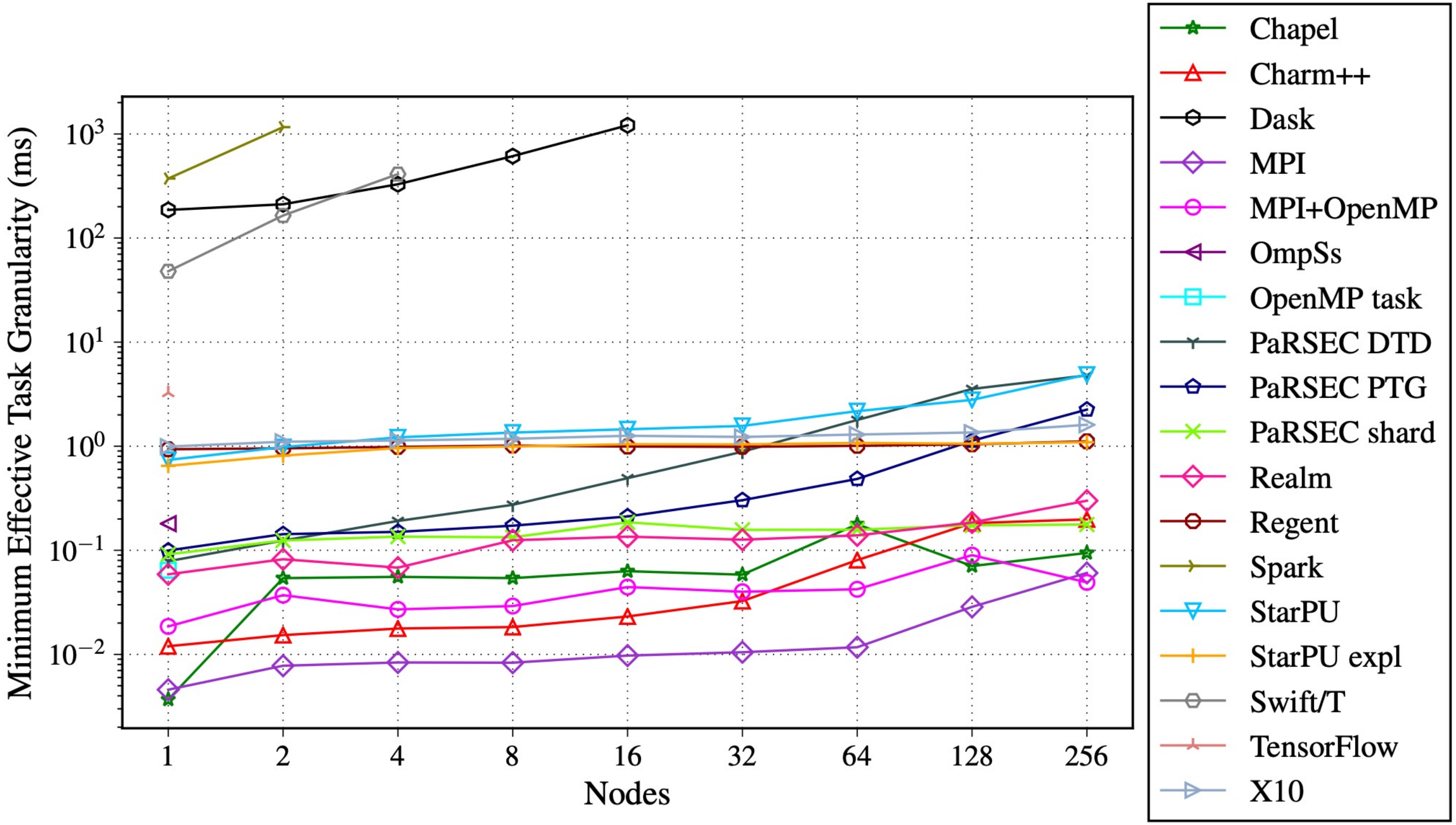
Who Would Switch from Big Data to HPC?

0%

Who Would Switch from HPC to Big Data?

- If performance improved by switching, everyone
- If performance were comparable or not overly harmed, some
- If performance is 10X worse, none
    - And some would not switch even if performance is only 2X worse

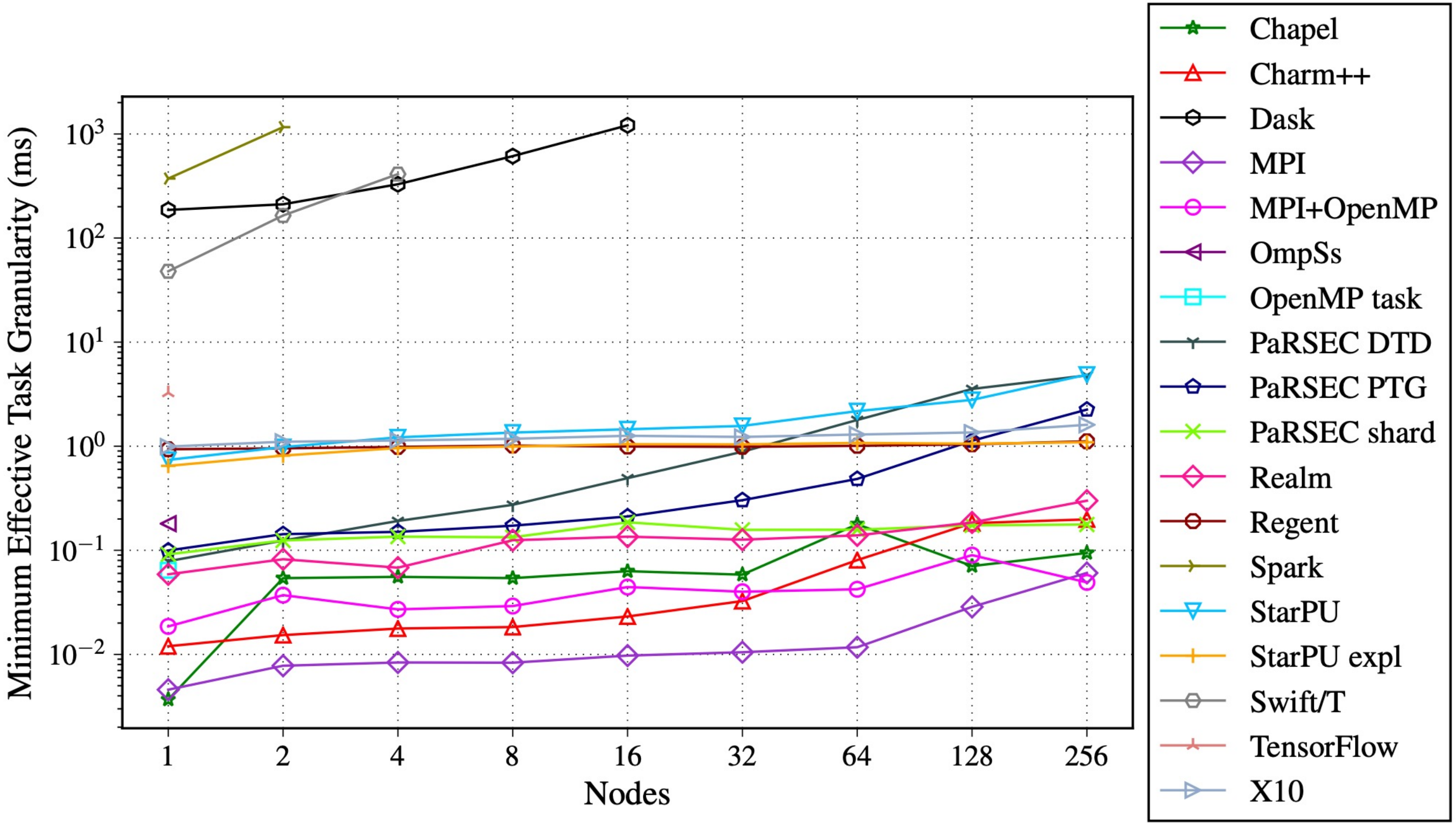
A Comparison: Minimum Task Granularity



(a) Stencil pattern.

Task Bench: A Parameterized Benchmark for Evaluating Parallel Runtime Performance, Slaughter et al, SC'20

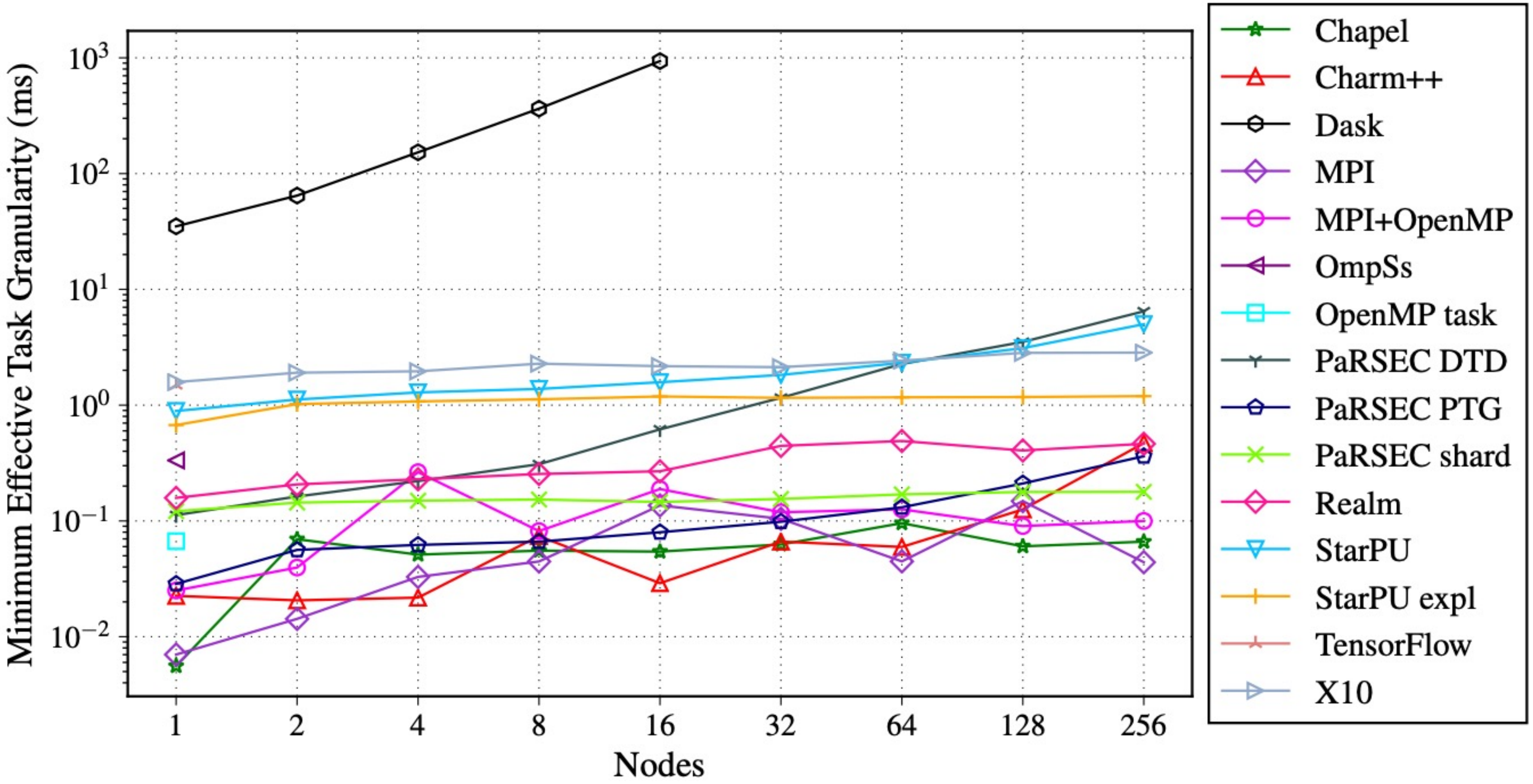
A Comparison: Minimum Task Granularity



(a) Stencil pattern.

Task Bench: A Parameterized Benchmark for Evaluating Parallel Runtime Performance, Slaughter et al, SC'20

A Comparison: Minimum Task Granularity



(d) Nearest pattern, 5 deps/task, 4 independent graphs.

Task Bench: A Parameterized Benchmark for Evaluating Parallel Runtime Performance, Slaughter et al, SC’20

A Brief Digression: Hardware

- The hardware platform drives the software abstractions

- The current, slow-motion revolution: accelerators
  - GPUs today
  - Other specialized hardware tomorrow

A Key Point

- In new supercomputers, > 95% of performance is in the accelerators
  - Titan, Summit, PerlMutter, Frontier, Aurora …

- The tradeoff
  - Greatly complicates programming
  - But switching to GPUs can greatly increase performance

- This is the ground on which any convergence will happen

An Observation

- The HPC community values performance
  - Unless it is too hard
  - Many HPC systems perform far below their potential today

- The Big Data community values productivity
  - Until the code takes forever to run
  - Organizations spend inordinate amounts of time tweaking for performance

The Technical Issue

- The main limiter in current and future systems is data movement
  - By far the most expensive part of any computation
  - And accelerators add multiple levels of memory hierarchy

- Few programming abstractions in programming models for
  - Locality
  - Partitioning of data
  - Mapping of compute/data into a machine

The Evidence

- S3D
  - Production chemistry combustion code
  - 7X off its potential
- Large graph analytics
  - CPU-based state of the art ~10X off potential

Switching to GPUs + good data partitioning & placement

- Machine Learning
  - 10X off potential

Improved data partitioning

Where Does Productivity Come From?

- Libraries
- How many widely used parallel libraries for HPC are there?
- How many widely used libraries are there for Python?
  - Not just “big data”

Numpy
In One Slide

A popular Python package for (mostly) dense array computing

Common building block in other Python packages

Many drop-in replacements for one GPU

```
import numpy as np

def cg_solve(A, b, tol=1e-10):
    x = np.zeros(A.shape[1])
    r = b - A.dot(x)
    p = r
    rsold = r.dot(r)
    for i in xrange(b.shape[0]):
        Ap = A.dot(p)
        alpha = rsold / (p.dot(Ap))
        x = x + alpha * p
        r = r - alpha * Ap
        rsnew = r.dot(r)
        if np.sqrt(rsnew) < tol:
           break
        beta = rsnew / rsold
        p = r + beta * p
        rsold = rsnew
 return x
```

Legate Numpy

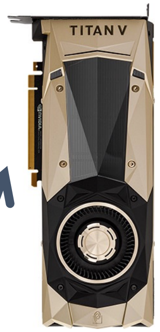
Accelerated and Distributed

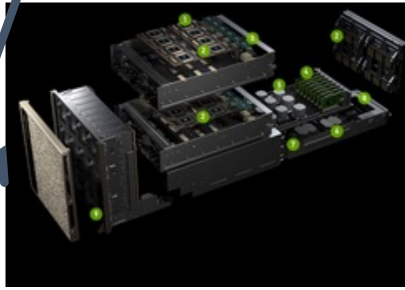
Legate NumPy is a NumPy replacement for transparent (weak) scaling

Requires a one line code change

Same code runs on everything

```
import legate.numpy as np

def cg_solve(A, b, tol=1e-10):
    x = np.zeros(A.shape[1])
    r = b - A.dot(x)
    p = r
    rsold = r.dot(r)
    for i in xrange(b.shape[0]):
        Ap = A.dot(p)
        alpha = rsold / (p.dot(Ap))
        x = x + alpha * p
        r = r - alpha * Ap
        rsnew = r.dot(r)
        if np.sqrt(rsnew) < tol:
           break
        beta = rsnew / rsold
        p = r + beta * p
        rsold = rsnew
    return x
```

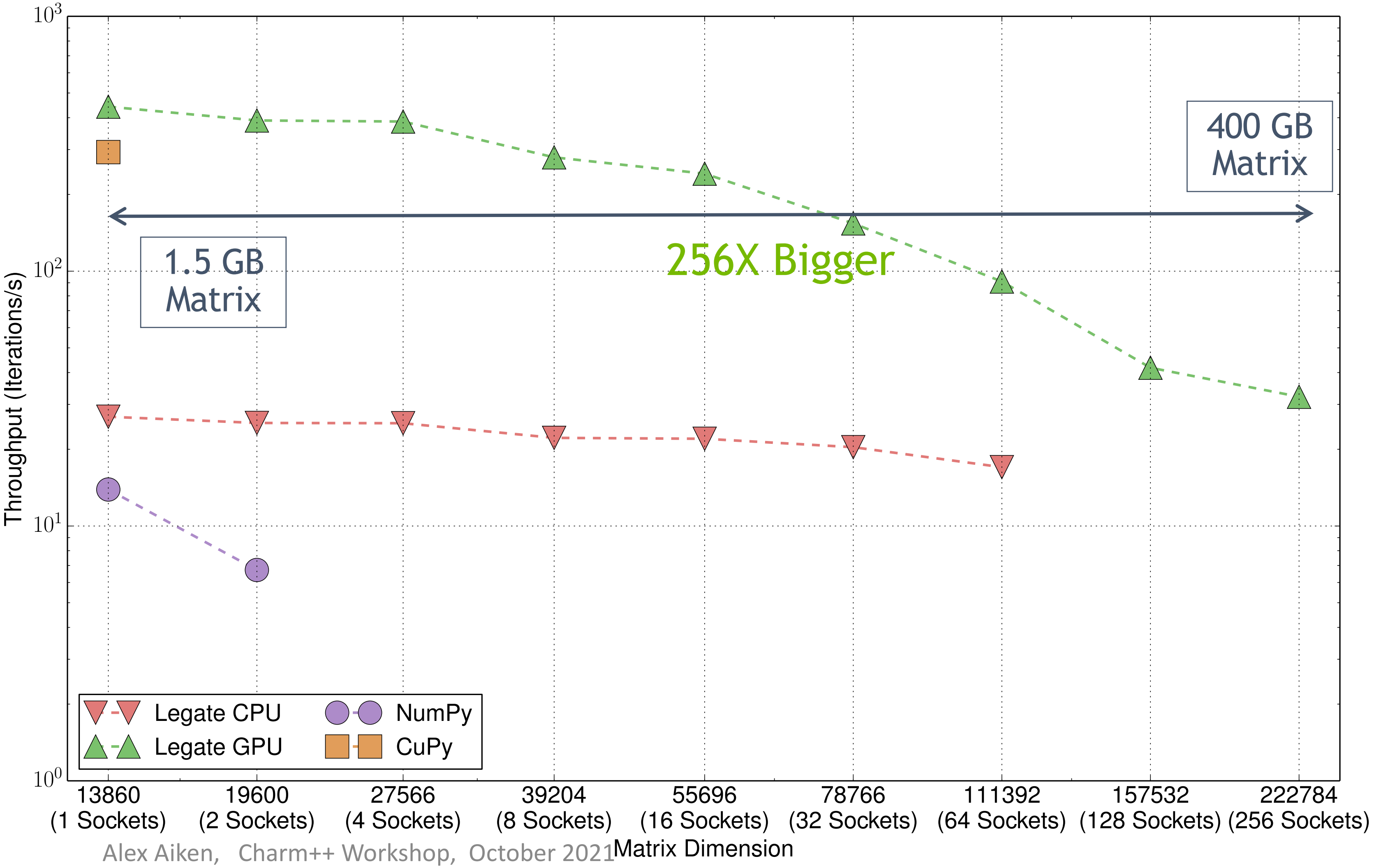




Legate NumPy: Accelerated and distributed array computing,   Bauer & Garland SC'19

A Simple Example: A Jacobi Solver

```
import legate.numpy as np

A = np.random.rand(N,N)
b = np.random.rand(N)

x = np.zeros(A.shape[1])
d = np.diag(A)
R = A - np.diag(d)
for i in xrange(b.shape[0]):
  x = (b - np.dot(R,x)) / d
```



Legate NumPy Architecture

Legate NumPy translates
API calls into task launches

Application

Program order

np.argmin

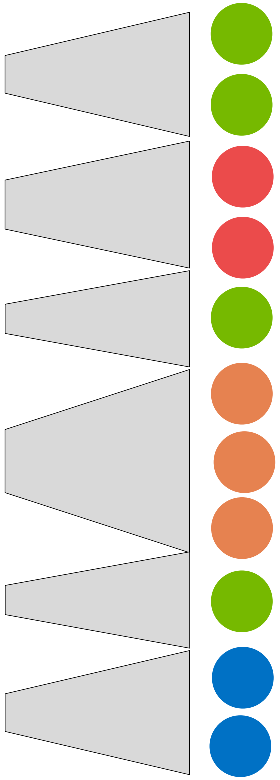
np.sort

np.add

np.dot

np.mul

np.norm



Legate NumPy provides
fast task implementations

C/C++ NVIDIA CUDA OpenMP

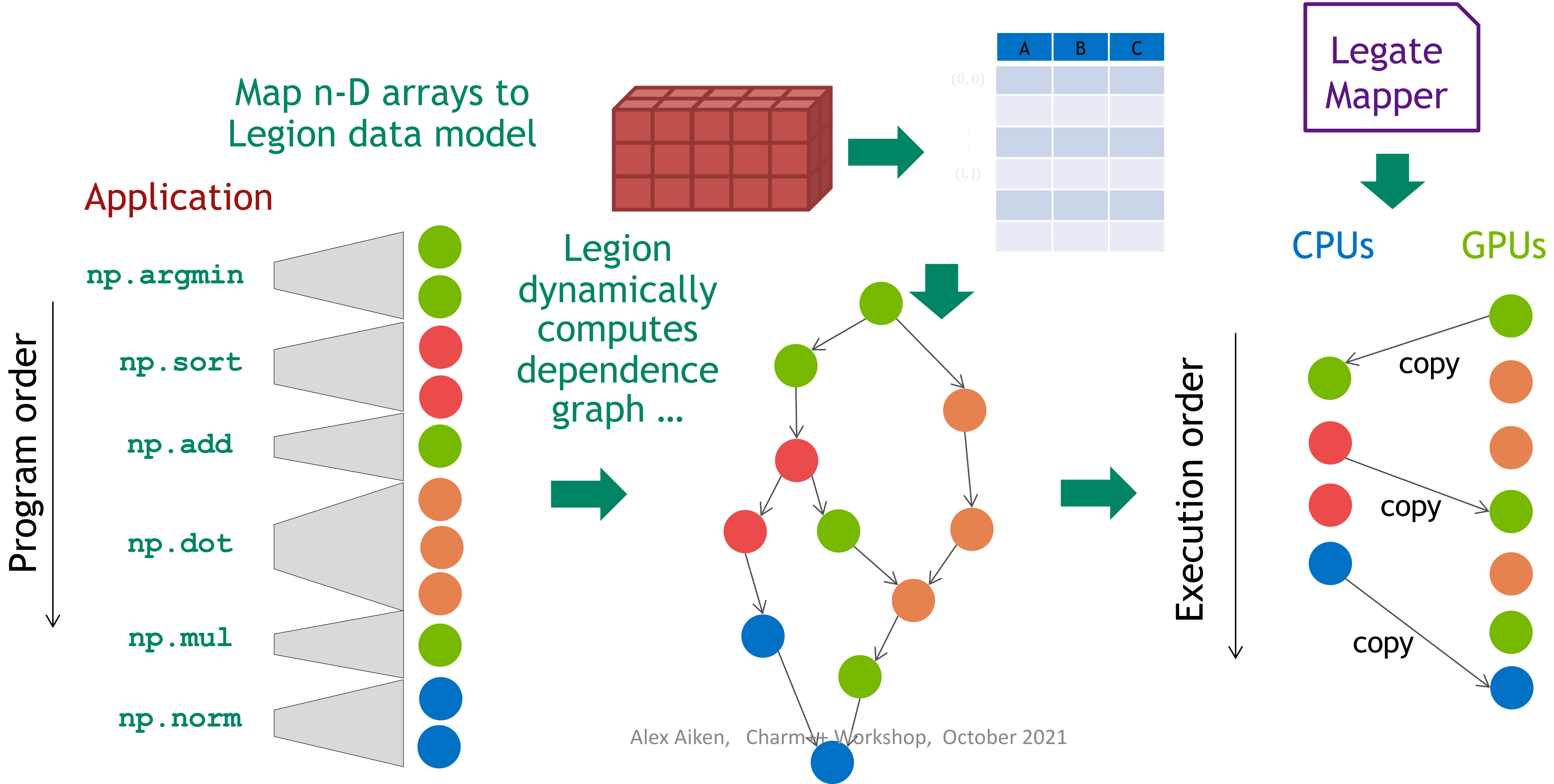
Legion

Data-Driven
Task-Based
Runtime

Legate NumPy provides a
custom implementation of
the Legion mapping interface

Legate
Numpy
Mapper

Legate NumPy architecture



Managing Data

Each N-D array maps to a field of a Legion *logical region*

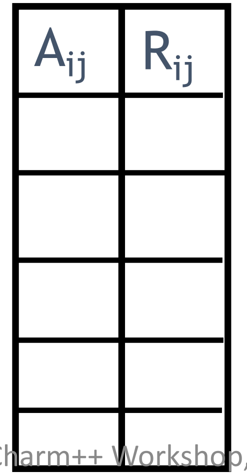
- Legion's collection data type

Different logical regions for different shapes

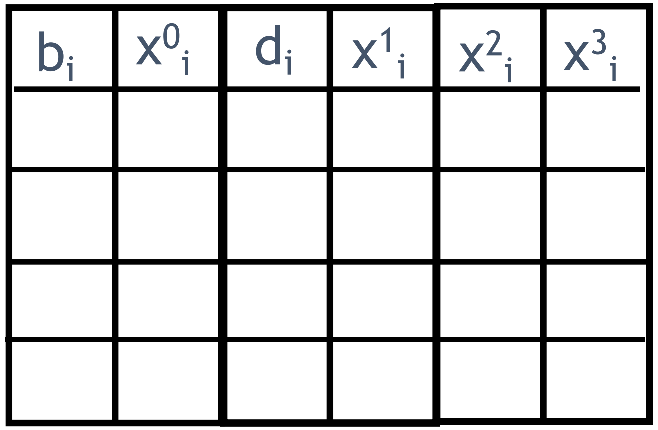
Dynamically allocated on demand and recycled when GC'd by Python

```
import legate.numpy as np

A = np.random.rand(N,N)
b = np.random.rand(N)

x = np.zeros(A.shape[1])
d = np.diag(A)
R = A - np.diag(d)
for i in xrange(b.shape[0]):
  x = (b - np.dot(R,x)) / d
```

Region with Index Space (N,N)



Region with Index Space (N)



Performance Comparison

Compare NumPy implementations:

Standard NumPy (single node)

IntelPy with MKL (single node)

Legate CPU-only

Legate CPU+GPU

Dask (CPU-only): Auto and Tuned

All plots are log-log

Experiments on a cluster of DGX-1V nodes

Weak scaling throughput on sockets

DASK

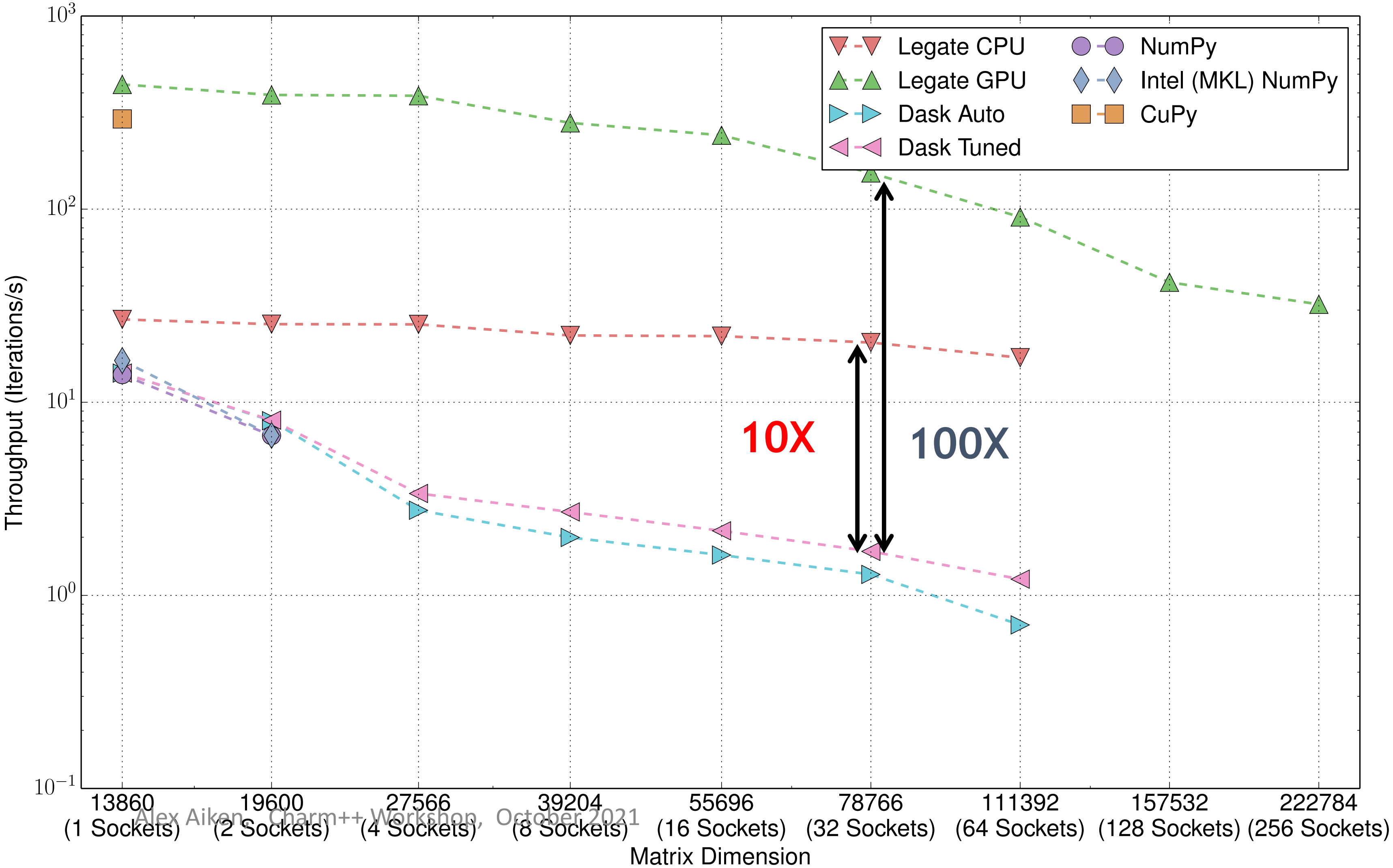
Popular Python library for parallel and distributed computing

dask.array similar to NumPy, except for specifying “chunk” sizes

```
import dask.array as da

A = da.random.uniform((N,N),
      chunks=(C,C))
b = da.random.uniform(N,
      chunks=“auto”)
x = da.zeros(A.shape[1],
      chunks=b.chunks)
d = da.diag(A)
R = A - da.diag(d)
for i in xrange(b.shape[0]):
  x = (b - da.dot(R,x)) / d
```

Jacobi Solver

```
import numpy as np

A = np.random.rand(N,N)
b = np.random.rand(N)

x = np.zeros(A.shape[1])
d = np.diag(A)
R = A - np.diag(d)
for i in xrange(b.shape[0]):
  x = (b - np.dot(R,x)) / d
```



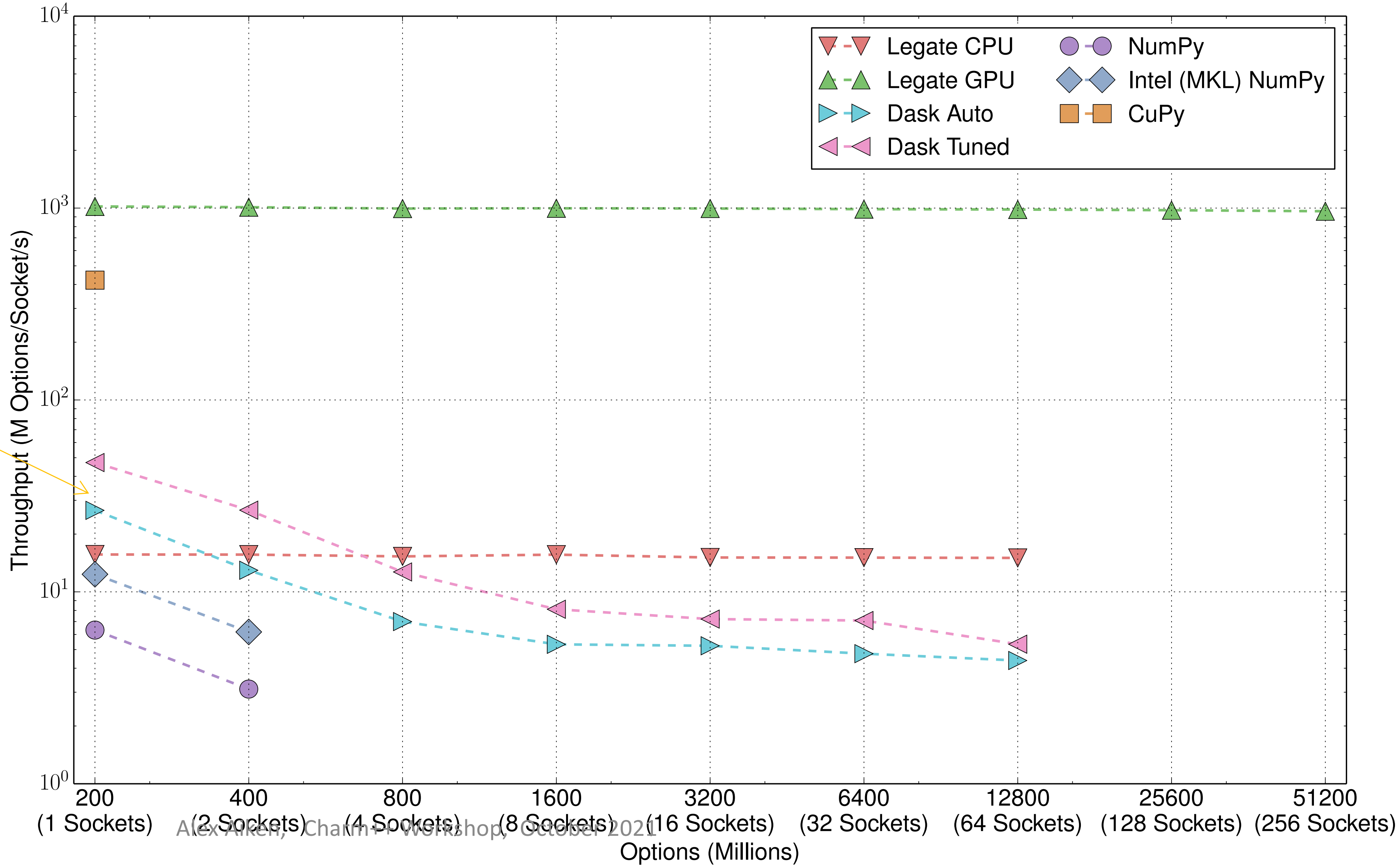
Black Scholes

No (application) communication

Expect perfect weak scaling

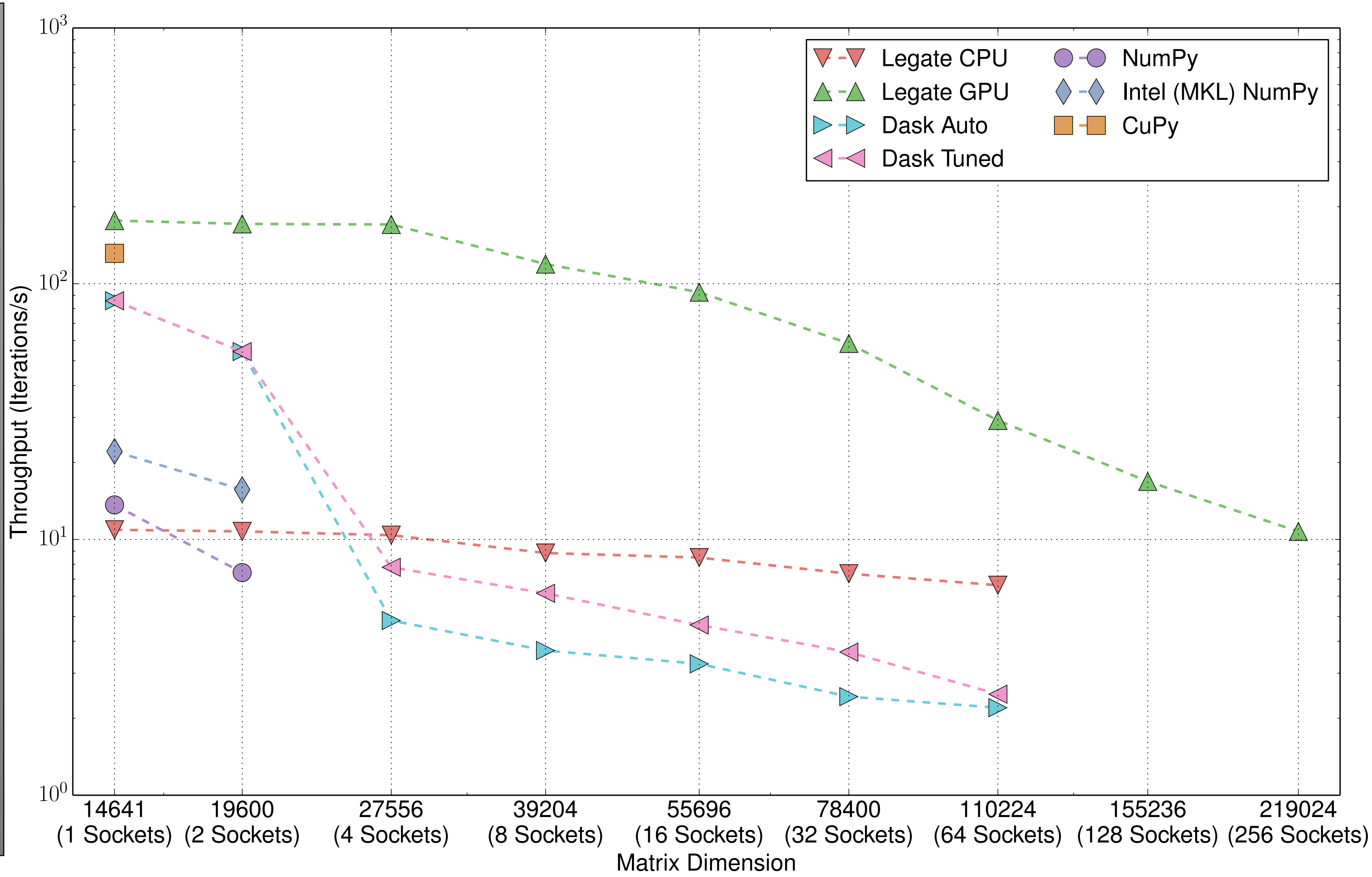
Dask starts out faster…
Why? Operator Fusion

… but has to trade off parallelism for task granularity to scale



Preconditioned CG Solver

```
def preconditioned_solve(A, M, b):
    x = np.zeros(A.shape[1])
    r = b - A.dot(x)
    z = M.dot(r)
    p = z
    rzold = r.dot(z)
    for i in xrange(b.shape[0]):
        Ap = A.dot(p)
        alpha = rzold / (p.dot(Ap))
        x = x + alpha * p
        r = r - alpha * Ap
        rznew = r.dot(r)
        if np.sqrt(rznew) < 1e-10:
            break
        z = M.dot(r)
        rznew = r.dot(z)
        beta = rznew / rzold
        p = z + beta * p
        rzold = rznew
    return x
```



One Approach To Libraries

- Implement important Big Data libraries using HPC techniques
  - Can we get more performance for the same productivity?

- Examples
  - Legate
  - FlexFlow, replacement for TensorFlow & PyTorch

    Beyond data and model parallelism for deep neural networks, Jia et al. SysML `18

Important Features

- Expressive data partitioning
- Ability to tune the *mapping*
  - Tasks to processors
  - Data to memories
- Runtime decision making
  - Needed to handle dynamic nature of Python
- Legion is extreme in all three dimensions
  - Sufficient, but maybe not necessary?

Another Approach

- Demonstrate the ability to build general libraries for HPC applications
  - That compete with the best-of-class HPC implementations
  - But are more productive to write and/or use

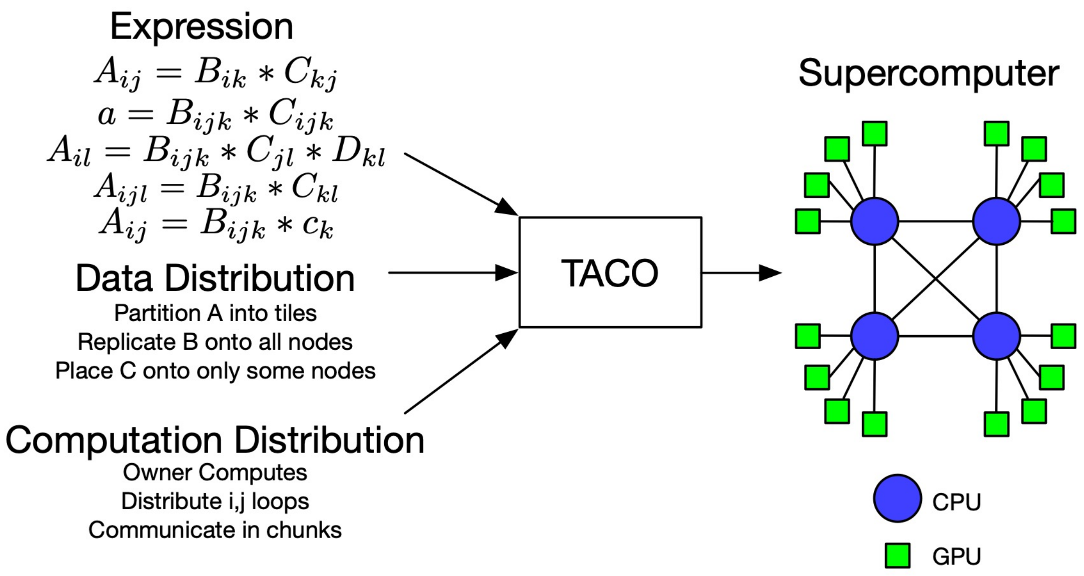
- What are the important/novel problems in building HPC libraries?

DISTAL: DIStributed Tensor ALgebra

Goals:

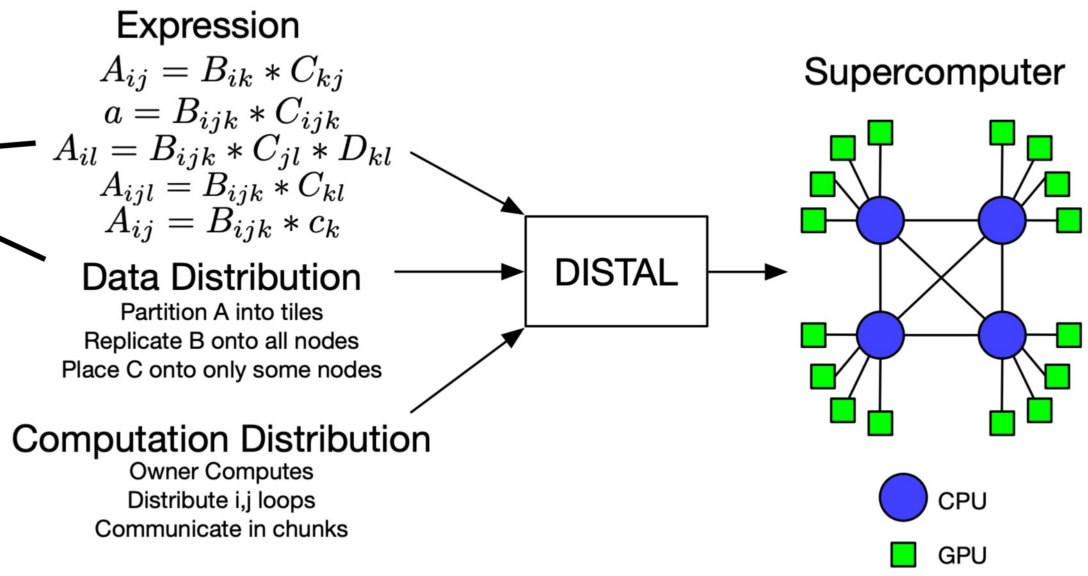
Compile tensor algebra kernels into efficient distributed implementations

Decouple computation, performance optimizations, and data distribution



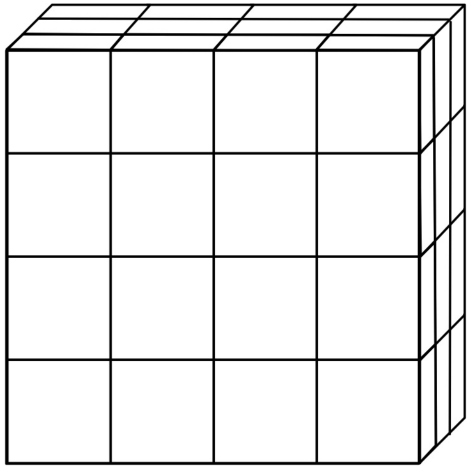
Joint work with Rohan Yadav and Fred Kjolstad

```
Param gx, gy, n;
Machine m(Grid(gx, gy));

Distribution tiles(m, {0, 1});
Format f({dense, dense}, tiles);
Tensor<double> a({n, n}, f), b({n, n}, f), c({n, n}, f);

IndexVar i, j, k;
a(i, j) = b(i, k) * c(k, j);

IndexVar in, jn, il, jl, ko, ki;
a.schedule()
 .divide(i, in, il, m.x).divide(j, jn, jl, m.y).divide(k, ko, ki, m.x)
 .reorder({in, jn, il, jl})
 .distribute({in, jn}, DistributedGPU)
 .reorder({ko, il, il, ki})
 .communicate(a, jn).communicate({b, c}, ko)
 .substitute({il, jl, ki}, CuBLAS::GeMM)
 ;

a.compile();
```



Modeling Machines

- View machines as hyper-rectangular grids of processors
  - where each processor has a local memory
- Expose any locality in the physical machine
- Structure the machine like the target computations



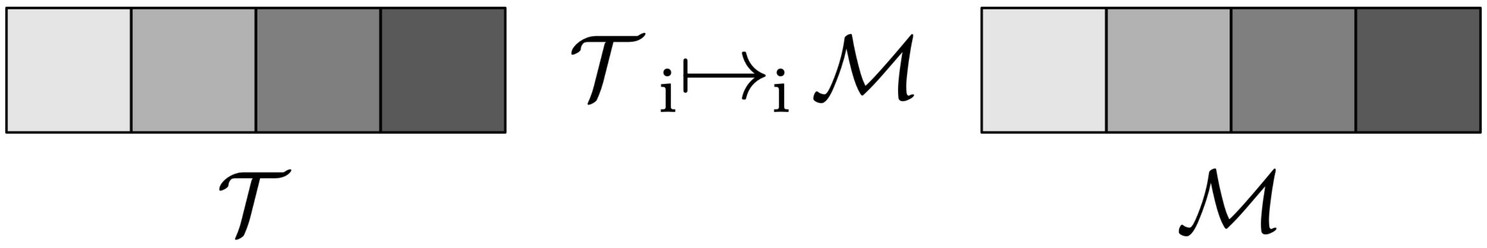
Distributing Data

- State abstractly how a tensor is distributed onto a machine as part of the tensor's *format*
- Describes how dimensions of a tensor $\mathcal{T}$ map onto a machine $\mathcal{M}$

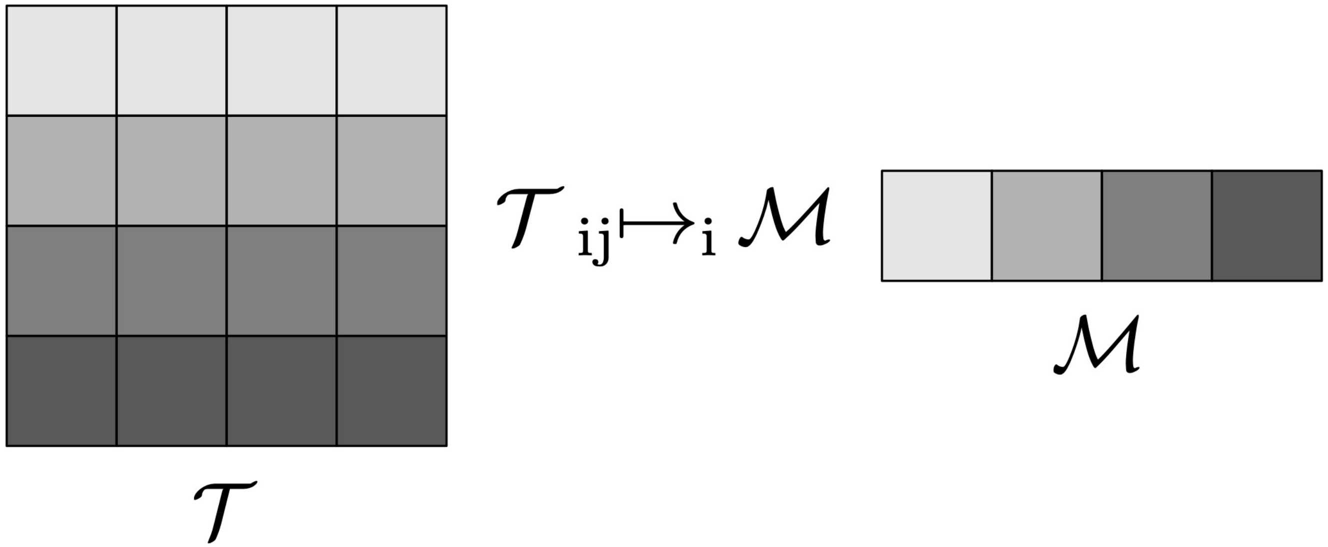
$$\mathcal{T}\ {}_{\mathrm{ij}}\!\mapsto_{\mathrm{i}} \mathcal{M}$$

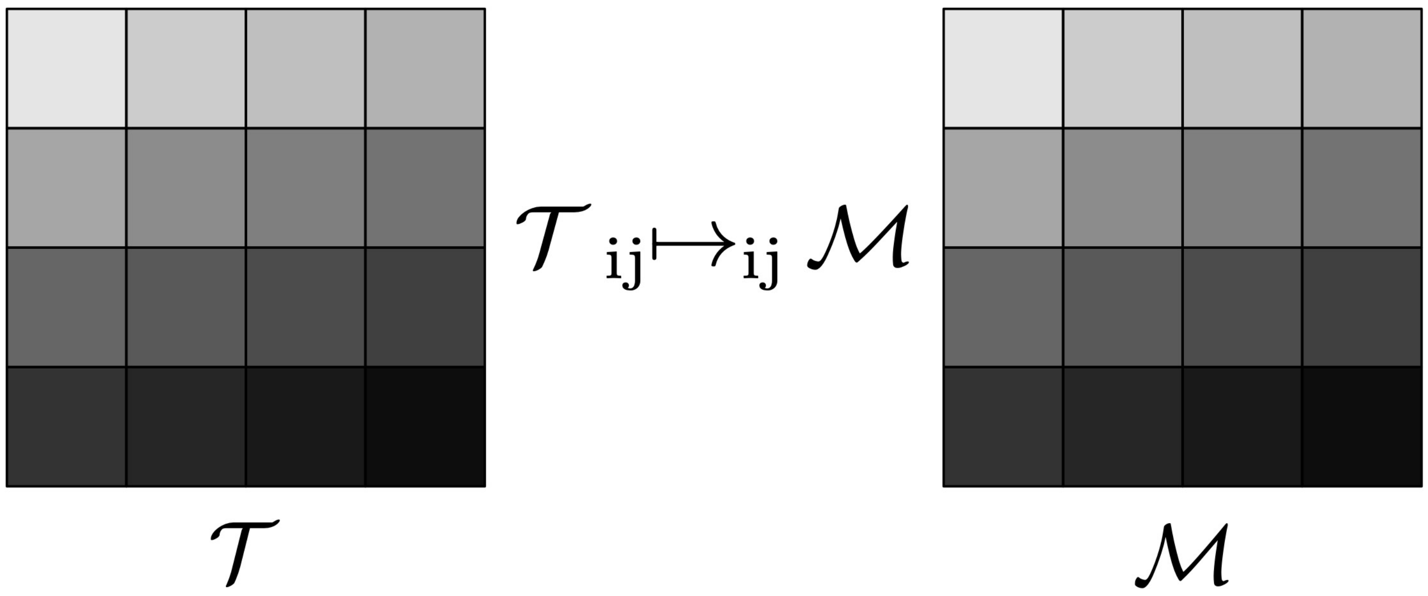
Name each dimension of $\mathcal{T}$ and $\mathcal{M}$

Dimensions of $\mathcal{T}$ are partitioned and mapped onto dimensions of $\mathcal{M}$ that share the same name

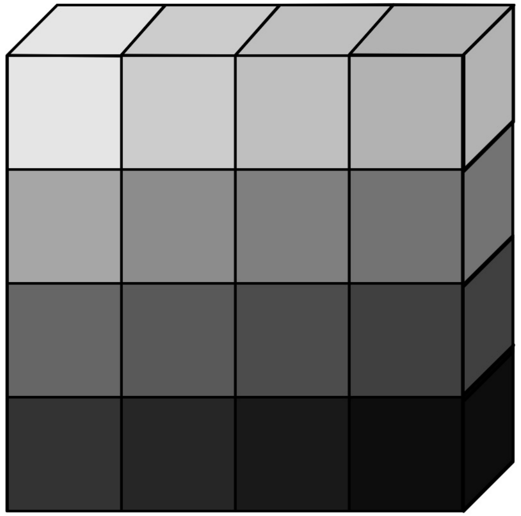


$\mathcal{T}$
${}_{\mathrm{i}}\!\mapsto_{\mathrm{i}}$
$\mathcal{M}$
$\mathcal{T}$
$\mathcal{M}$


$\mathcal{T}$
$\mathcal{T}\ {}_{\mathrm{ij}\mapsto\mathrm{i}}\ \mathcal{M}$
$\mathcal{M}$

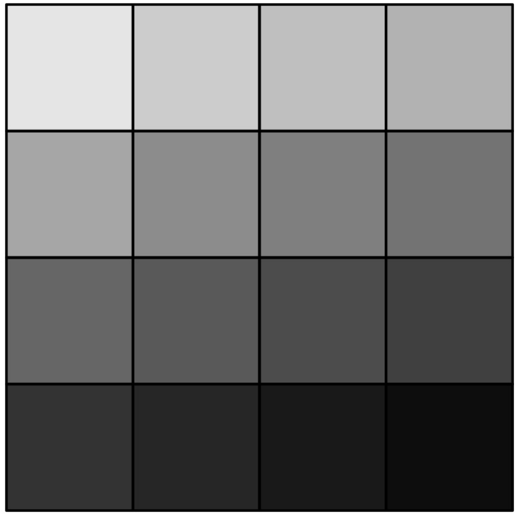


$\mathcal{T}\ {}_{\mathrm{ij}}{\mapsto}_{\mathrm{ij}}\ \mathcal{M}$
$\mathcal{T}$
$\mathcal{M}$

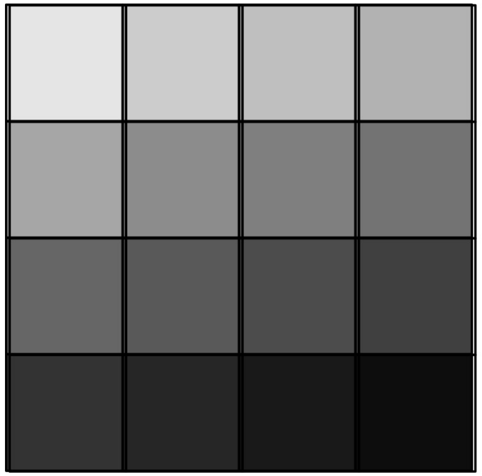


$\mathcal{T}$

$$\mathcal{T}_{\mathrm{ijk}} \mapsto_{\mathrm{ij}} \mathcal{M}$$

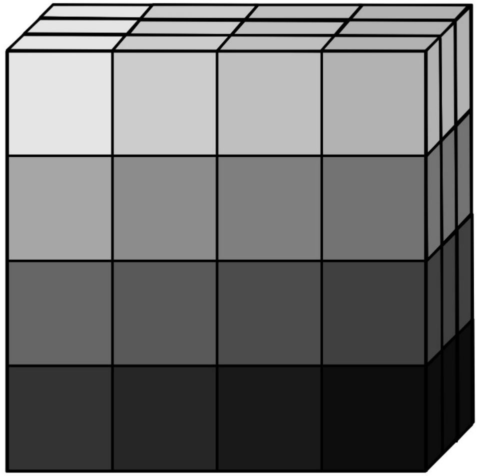


$\mathcal{M}$

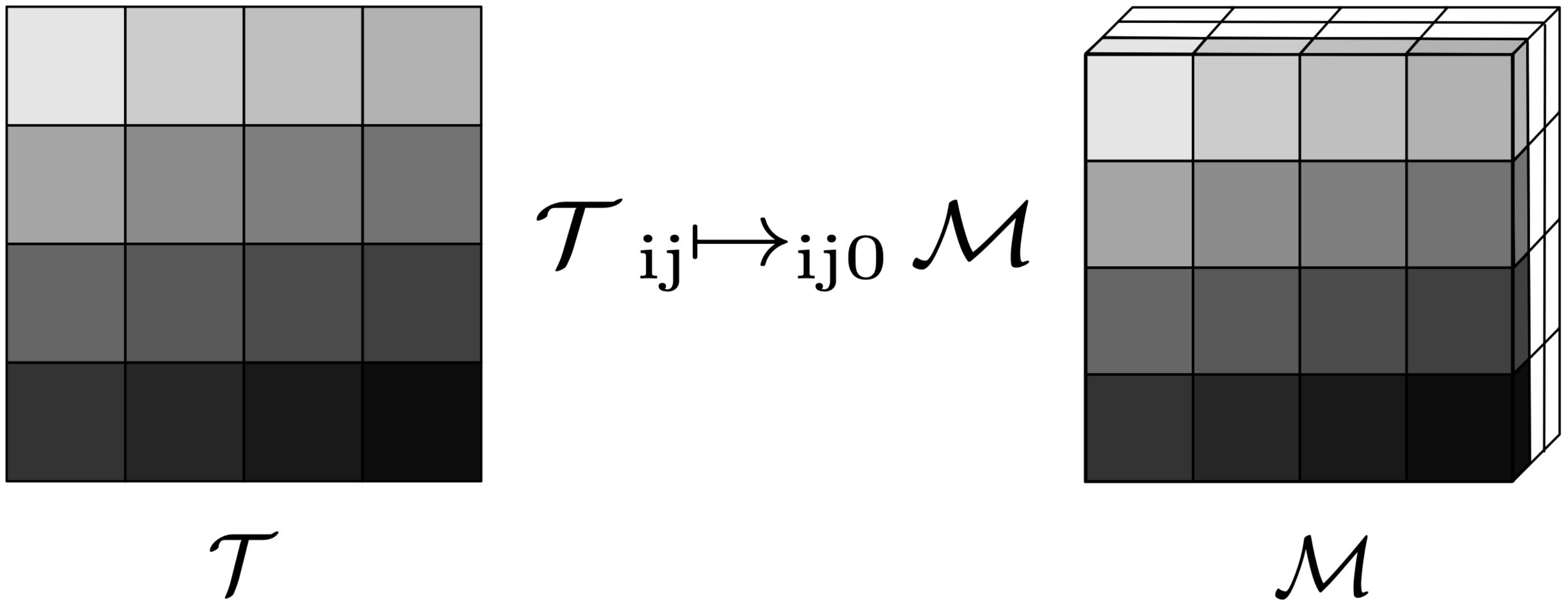


$\mathcal{T}$

$$\mathcal{T}_{ij} \mapsto_{ij*} \mathcal{M}$$



$\mathcal{M}$


$\mathcal{T}_{ij} \mapsto \mathcal{M}_{ij0}$
$\mathcal{T}$
$\mathcal{M}$

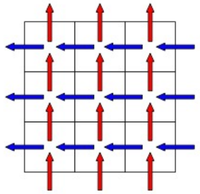
Scheduling (Summary)

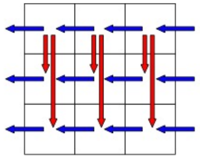
- Iteration spaces: hyper-rectangular grids representing points in nested loops

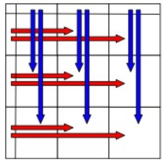
$$\forall_i \ A_i = \sum_j B_j$$

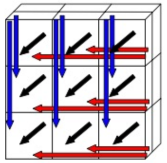
- Execution space: processors in $\mathcal{M}$ *x* time dimension
- Scheduling commands related to distribution change mapping of iteration space points to the execution space
- Apply scheduling commands to the computation
  - Similar to Halide schedules, with extensions for distributed computing
  - New commands: *distribute*, *communicate*, *rotate*

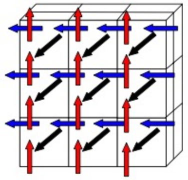
Algorithm	Comm. Pattern	Target Machine	Data Distribution	Schedule

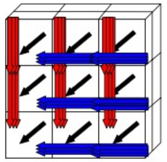
Cannon's [7]
(1969)
		$\mathcal{M}(gx, gy)$	$A_{ij \mapsto ij}\, \mathcal{M}$
			$B_{ij \mapsto ij}\, \mathcal{M}$
			$C_{ij \mapsto ij}\, \mathcal{M}$
				`.distribute({i, j}, {in, jn}, {il, jl}, Grid(gx, gy))`
				`.divide(k, ko, ki, gx)`
				`.reorder({ko, il, jl, ki})`
				`.rotate(ko, {in, jn}, kos)`
				`.communicate(A, jn)`
				`.communicate({B, C}, kos)`

PUMMA [10]
(1994)
		$\mathcal{M}(gx, gy)$	$A_{ij \mapsto ij}\, \mathcal{M}$
			$B_{ij \mapsto ij}\, \mathcal{M}$
			$C_{ij \mapsto ij}\, \mathcal{M}$
				`.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)`

SUMMA [25]
(1995)
		$\mathcal{M}(gx, gy)$	$A_{ij \mapsto ij}\, \mathcal{M}$
			$B_{ij \mapsto ij}\, \mathcal{M}$
			$C_{ij \mapsto ij}\, \mathcal{M}$
				`.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)`

Johnson's [1]
(1995)
		$\mathcal{M}(\sqrt[3]{p}, \sqrt[3]{p}, \sqrt[3]{p})$	$A_{ij \mapsto ij0}\, \mathcal{M}$
			$B_{ik \mapsto i0k}\, \mathcal{M}$
			$C_{kj \mapsto 0jk}\, \mathcal{M}$
				`.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)`

Solomonik's [22]
(2011)
		$\mathcal{M}(\sqrt{\frac{p}{c}}, \sqrt{\frac{p}{c}}, c)$	$A_{ij \mapsto ij0}\, \mathcal{M}$
			$B_{ij \mapsto ij0}\, \mathcal{M}$
			$C_{ij \mapsto ij0}\, \mathcal{M}$
				`.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)`

COSMA [17]
(2019)
		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)`

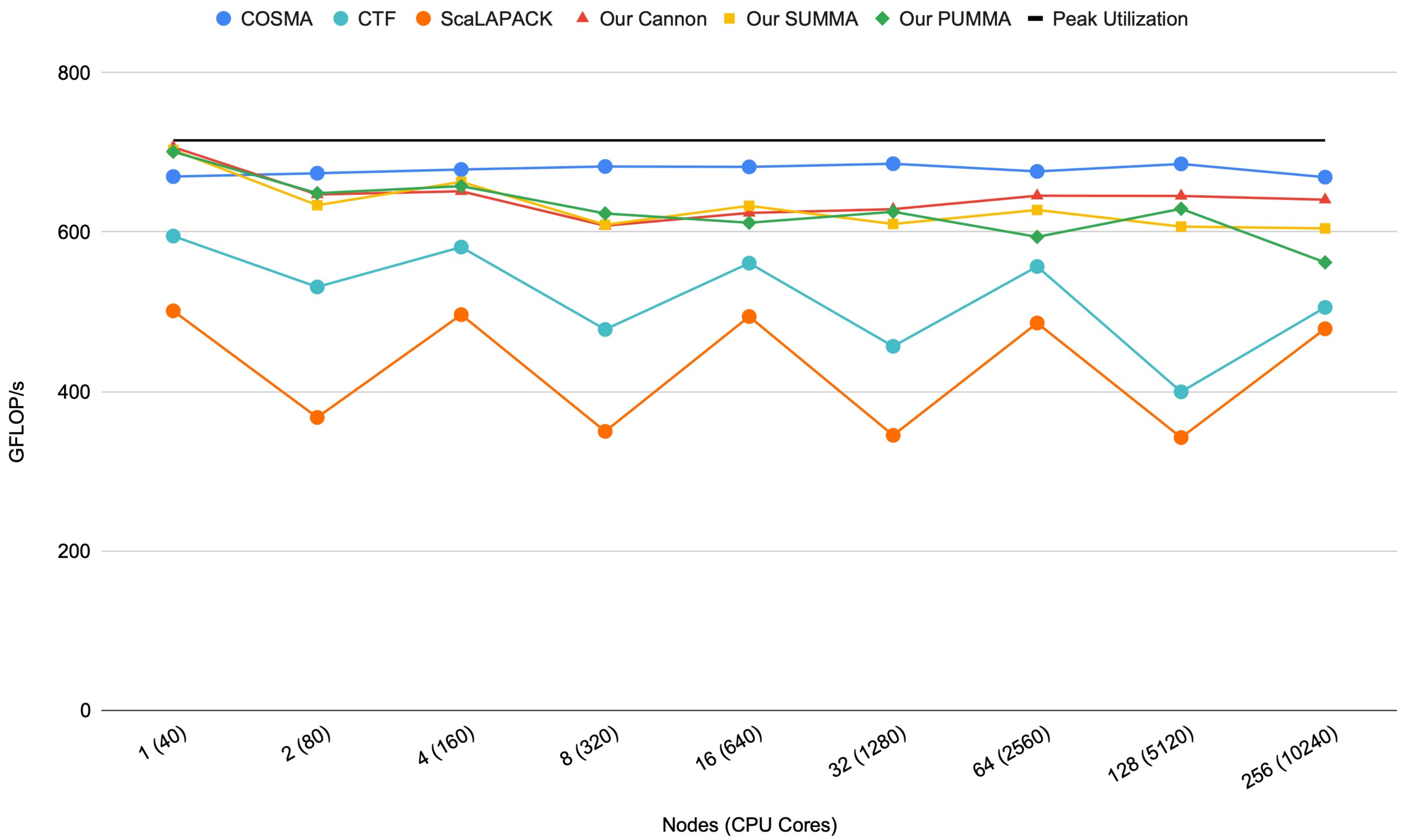
Experiments

- Run on Lassen
  - 4 GPUs/node, 40 CPUs/node, IB interconnect)

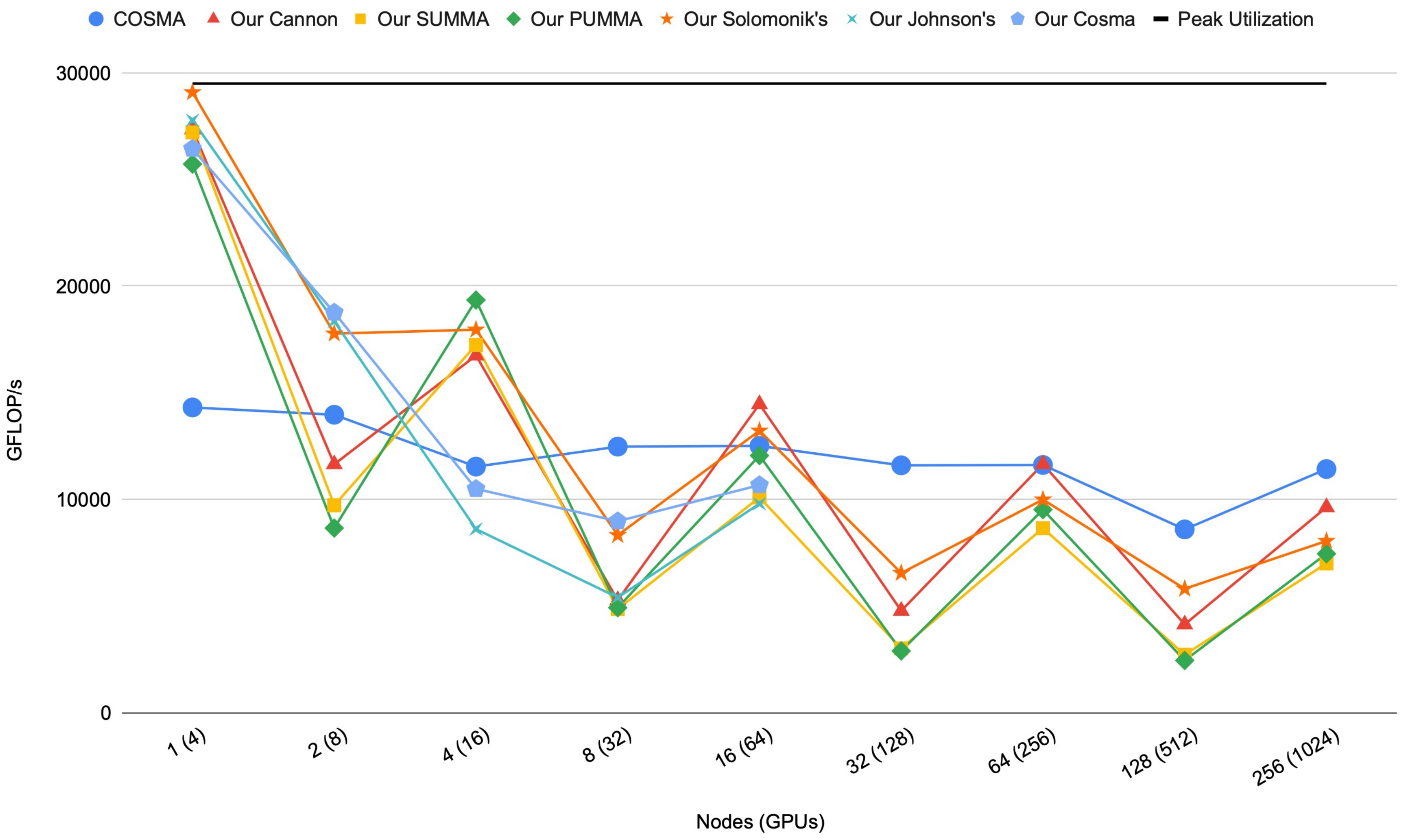
- All systems configured to use the same BLAS / CuBLAS

- All experiments are weak-scaling (memory / node stays constant)

GEMM (CPU)


COSMA
CTF
ScaLAPACK
Our Cannon
Our SUMMA
Our PUMMA
Peak Utilization
800
600
400
200
0
GFLOP/s
1 (40)
2 (80)
4 (160)
8 (320)
16 (640)
32 (1280)
64 (2560)
128 (5120)
256 (10240)
Nodes (CPU Cores)

GEMM (GPU)

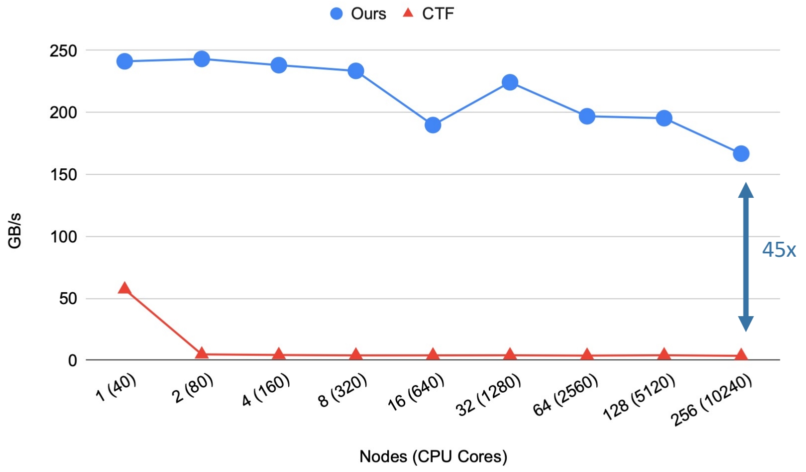


COSMA
Our Cannon
Our SUMMA
Our PUMMA
Our Solomonik's
Our Johnson's
Our Cosma
Peak Utilization
30000
20000
10000
0
GFLOP/s
1 (4)
2 (8)
4 (16)
8 (32)
16 (64)
32 (128)
64 (256)
128 (512)
256 (1024)
Nodes (GPUs)

Higher Order Tensor Operations (CPU)

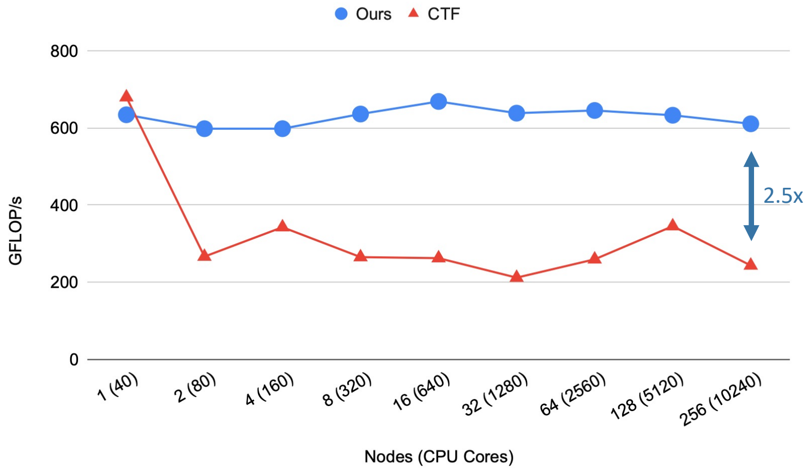
TTV

$$A_{ij} = B_{ijk} \cdot C_k$$



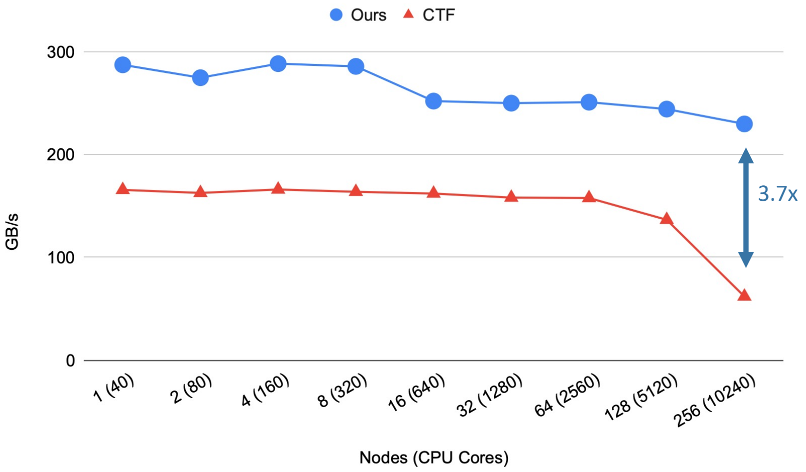
TTM

$$A_{ijl} = B_{ijk} \cdot C_{kl}$$



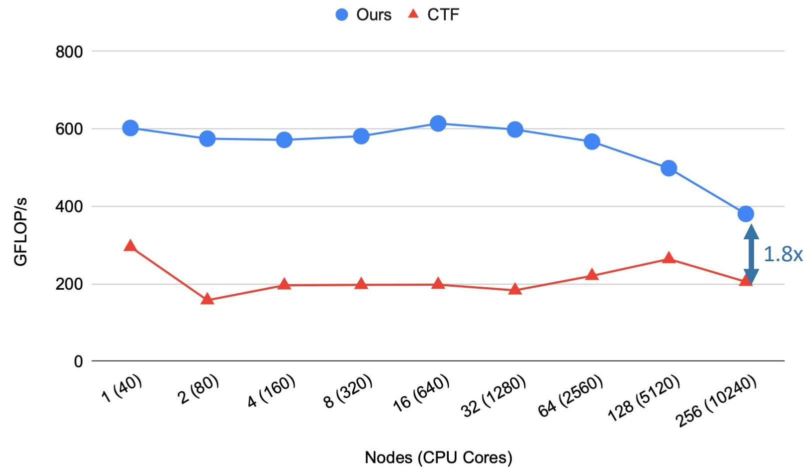
InnerProd

$$a = B_{ijk} \cdot C_{ijk}$$



MTTKRP

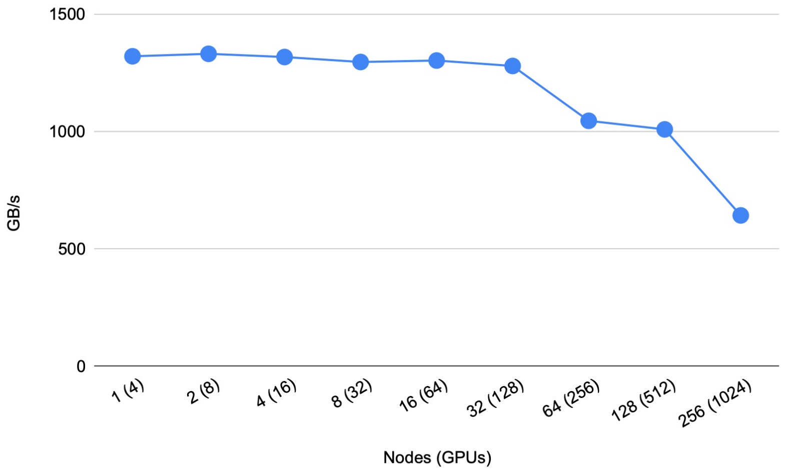
$$A_{il} = B_{ijk} \cdot C_{jl} \cdot D_{kl}$$



Higher Order Tensor Operations (GPU)

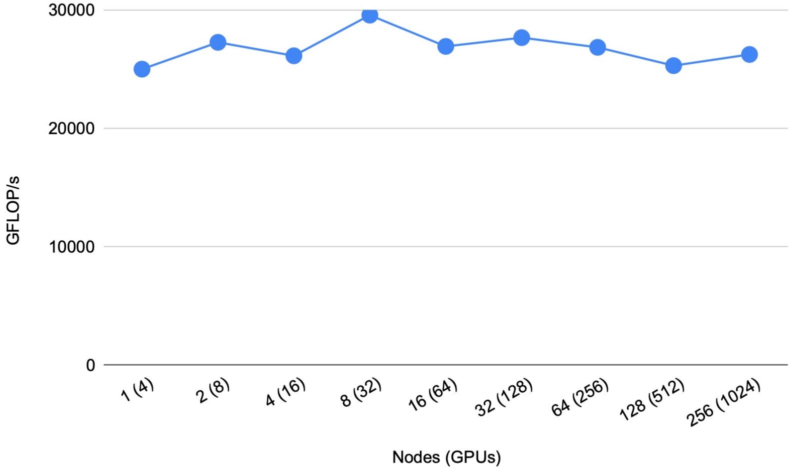
TTV

$$A_{ij} = B_{ijk} \cdot C_k$$



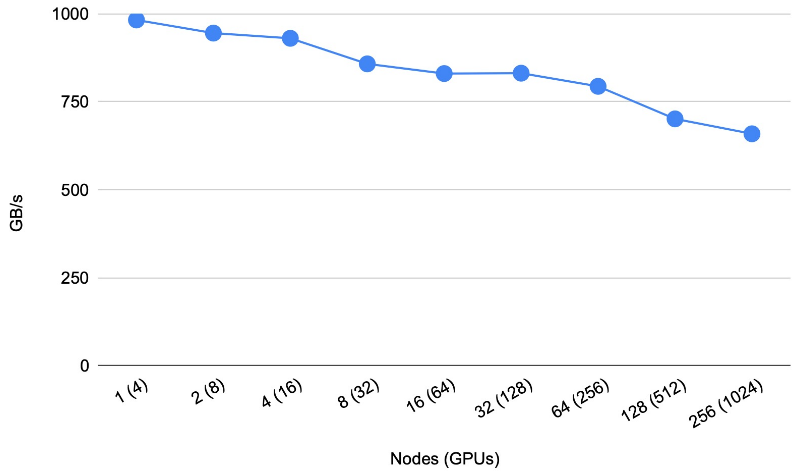
TTM

$$A_{ijl} = B_{ijk} \cdot C_{kl}$$



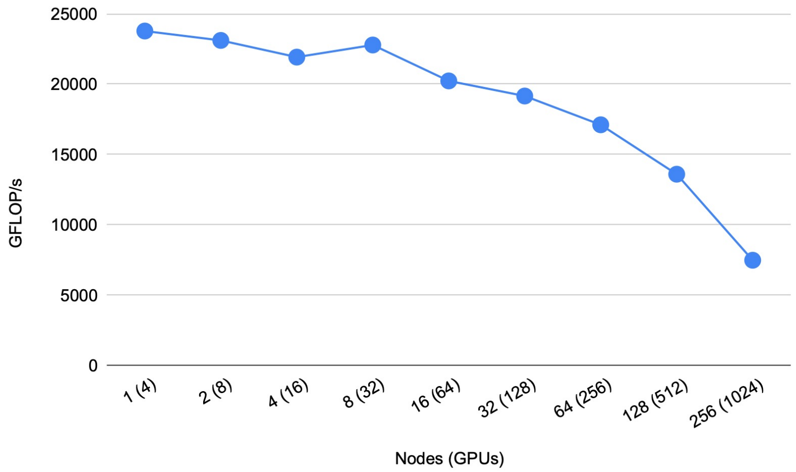
InnerProd

$$a = B_{ijk} \cdot C_{ijk}$$



MTTKRP

$$A_{il} = B_{ijk} \cdot C_{jl} \cdot D_{kl}$$



Lessons From DISTAL

- Expressive partitioning of data, computation and control of the mapping into the machine are all critical

- Enables writing libraries that are polymorphic in the data distribution
  - The data distribution can be different depending on the needs of the context
  - Avoids stopping-the-world and doing large copies at library boundaries
  - A form of polymorphism unique to distributed parallel programming

Summary

- The HPC and Big Data worlds have agreed on the hardware platform
  - Parallel, accelerated, distributed (PAD) machines
  - A convergence of these two worlds is likely

- Can we have both productivity and performance?
  - There is some preliminary evidence the answer is ``yes”
  - Through libraries built on HPC programming models
  - But libraries required a degree of flexibility beyond non-library code
    - Still much to be learned about how to write reusable parallel libraries