Task-Based Programming with Legion

Alex Aiken

Stanford/SLAC

What is Legion?

A task-based programming model for heterogeneous, parallel, distributed machines

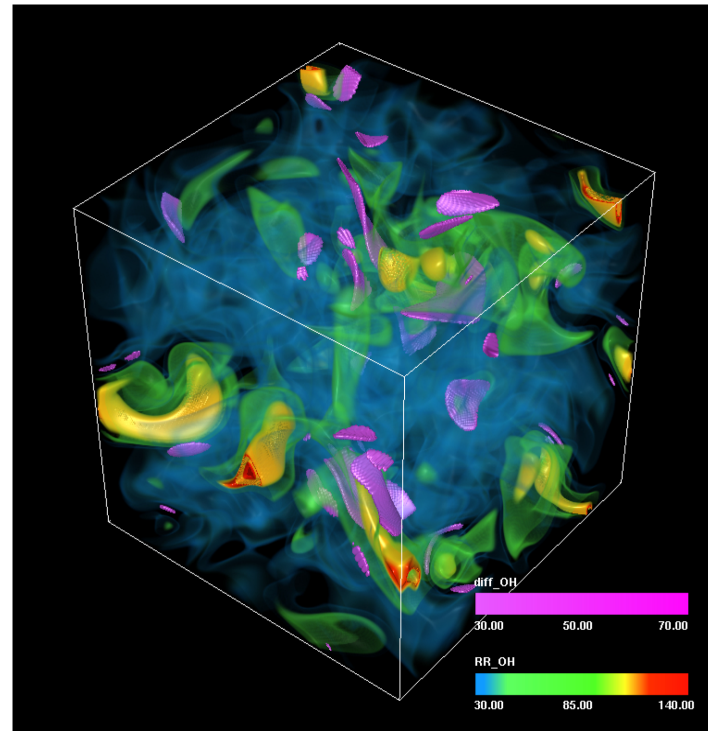
Designed to be

- High performance
- Performance portable
- Productive

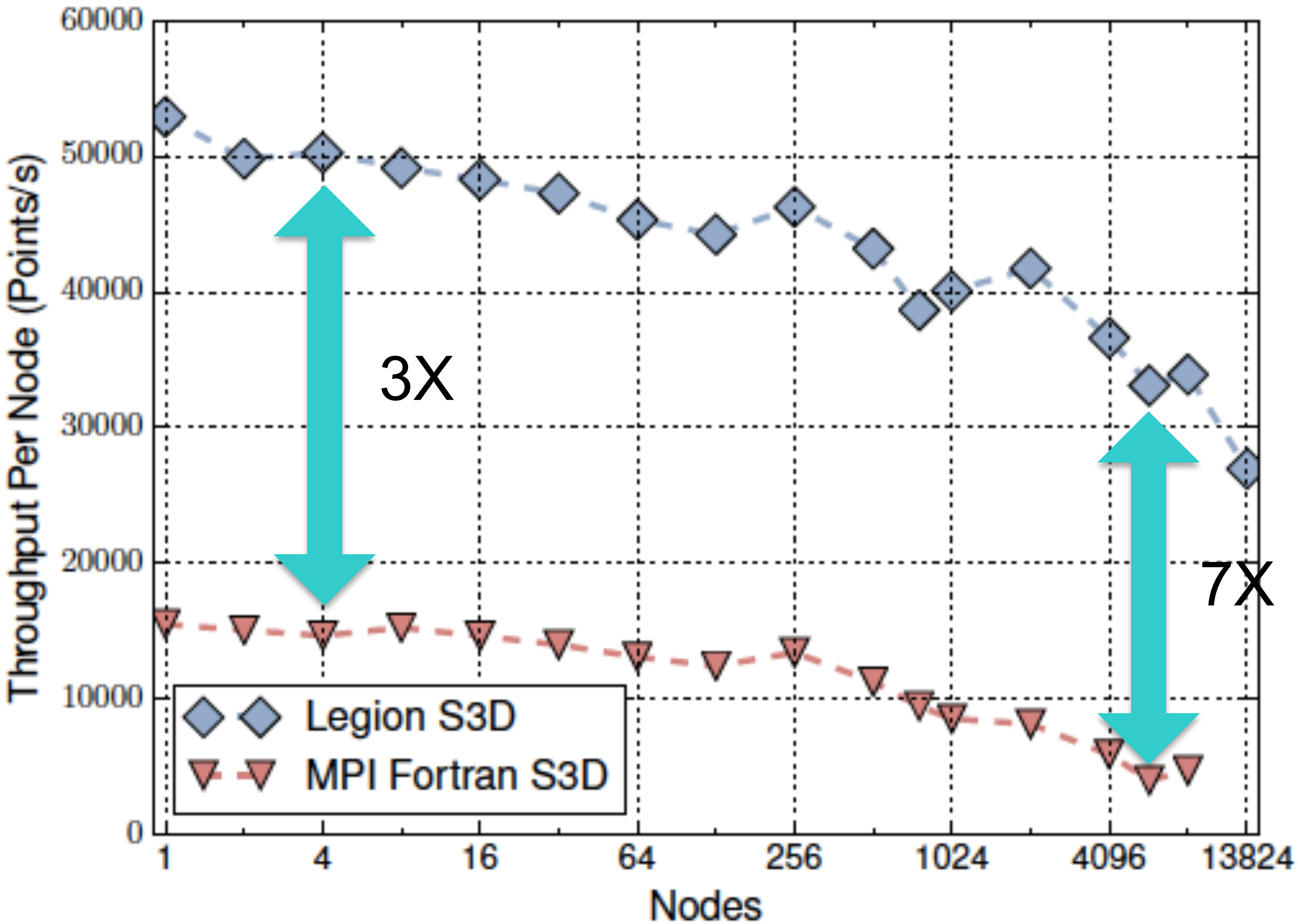
An Example: S3D

- Simulates chemical reactions
  - DME (30 species)
  - Heptane (52 species)
  - PRF (116 species)

- Two parts
  - Physics
    - Nearest neighbor communication
    - Data parallel
  - Chemistry
    - Local
    - Complex task parallelism



Weak Scaling: PRF



What Led to the Improvement?

- Sequential semantics
- Asynchronous tasks
- Late binding of performance decisions
  - Where tasks execute
  - Where data is placed
  - How data is partitioned
  - …

Sequential Semantics

S3D Skeleton

```
task top_level() {
    V = simulation volume
    P[N] = partition V
    G[N] = ghost cells of V
    repeat
        Chem(P[i]) for i = 1..N
        Phys(P[i],G[i]) for i = 1..N
    until done
}

task Chem(V) { … }
task Phys(V,G) { … }
```

- A sequential program
    - With a parallel execution
- Greatly simplifies debugging
    - No race conditions!
- Sequential semantics can be relaxed if desired
    - E.g., for reductions

Some Actual S3D Code …

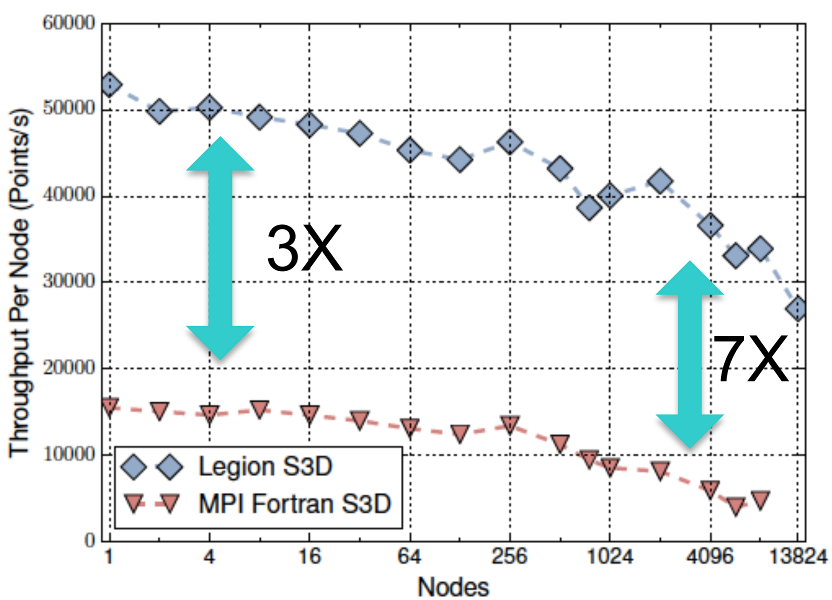
```
if compression() then
      __demand(__index_launch)
      for color in is_rank do
        CalcGammaTask(lp_int_rank[color])
      end

      __demand(__index_launch)
      for color in is_rank do
        Sum3Task(lp_int_rank[color].{X=RHS_1_DX, Y=RHS_1_DY, Z=RHS_1_DZ},
                 lp_q_rank[color].{RHO_U},
                 false)
      end

      __demand(__index_launch)
      for color in is_rank do
        Sum3Task(lp_int_rank[color].{X=RHS_2_DX, Y=RHS_2_DY, Z=RHS_2_DZ},
                 lp_q_rank[color].{RHO_V},
                 false)
      end

      __demand(__index_launch)
      for color in is_rank do
        Sum3Task(lp_int_rank[color].{X=RHS_3_DX, Y=RHS_3_DY, Z=RHS_3_DZ},
                 lp_q_rank[color].{RHO_W},
                 false)
      end
      …
```

Code is written in Regent.

Writing to the Legion C++ API has more details but the same structure.

The Benefits of Asynchrony

- Overlap communication and computation
- Overlap runtime analysis with the application
  - Runtime analysis is distributed SPMD fashion across nodes
- In general, also get task parallelism



Late Binding of Decisions

- After
  - the program is written
  - the machine is selected
  - the input is chosen
- It is easy to
  - Change the partitioning of data
  - Change the assignment of tasks
    - E.g., move a task from GPU to CPU
  - Change the placement of data
    - E.g., from the framebuffer to zero-copy memory
  - And more …

Mapping

```
Task * GPU,CPU;              # tasks run on GPUs by default

Task AwaitMPITask, CalcDummyTask, HandoffToMPITask, InitPartitionsTask, InitScaleTask, InitTemperatureTask, fill_cpe,
                             fill_lr_int, fill_masses CPU;

Region * * GPU FBMEM;        # for all GPU tasks, arguments use FBMEM as default
Region * * CPU SYSMEM;       # for CPU tasks, arguments use SYSMEM as default

Layout * * * SOA F_order;    # all regions use struct of array and Fortran order

…
```

The Secret Sauce

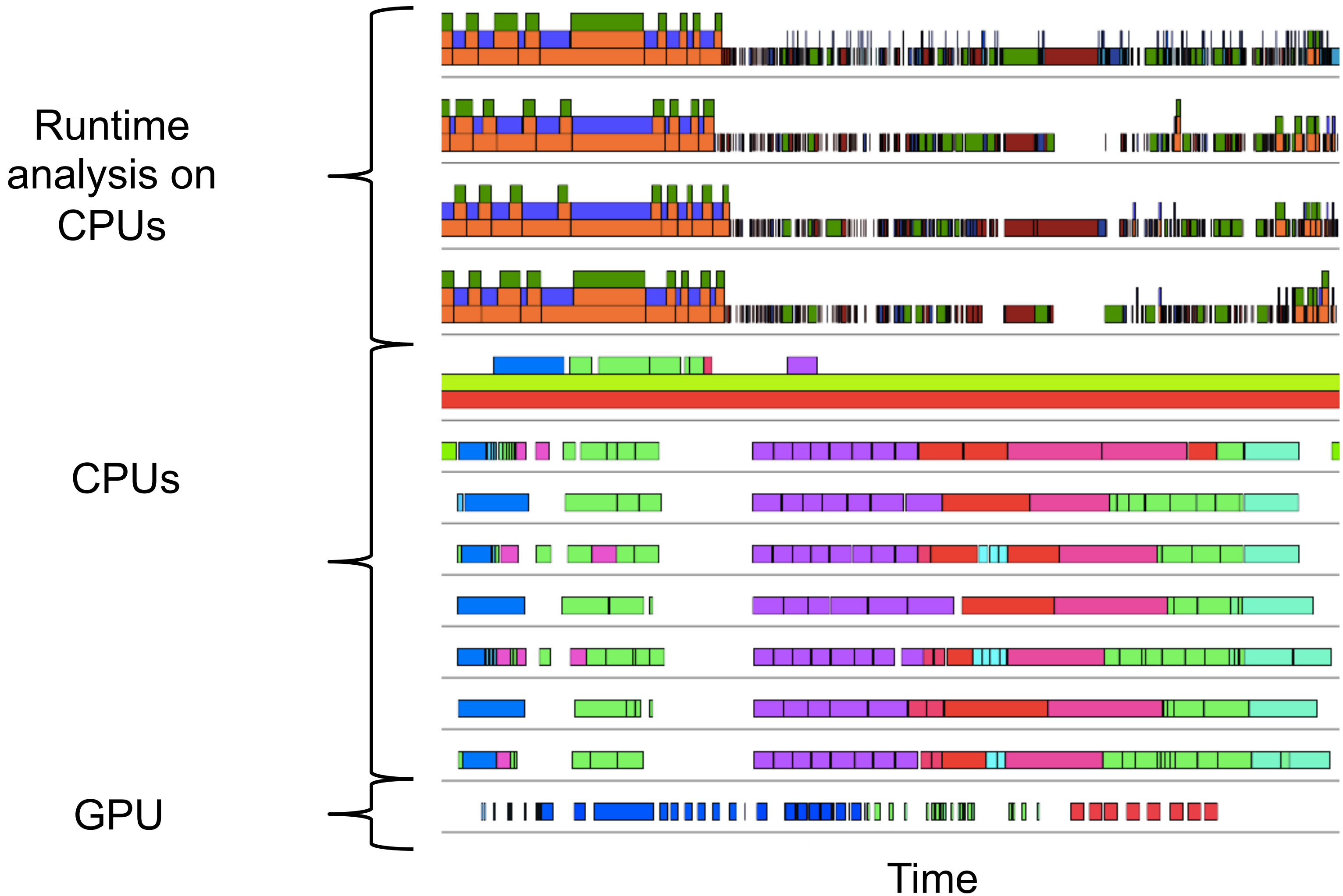
- The ability to easily change performance-relevant decisions after the program is running on a machine has been key
  - We often try a lot of different strategies!

- The biggest improvements of Legion over other approaches have not been because Legion's implementation strategy cannot be imitated.

- The improvements were because it was more productive to experiment in Legion to find an implementation strategy that works well.

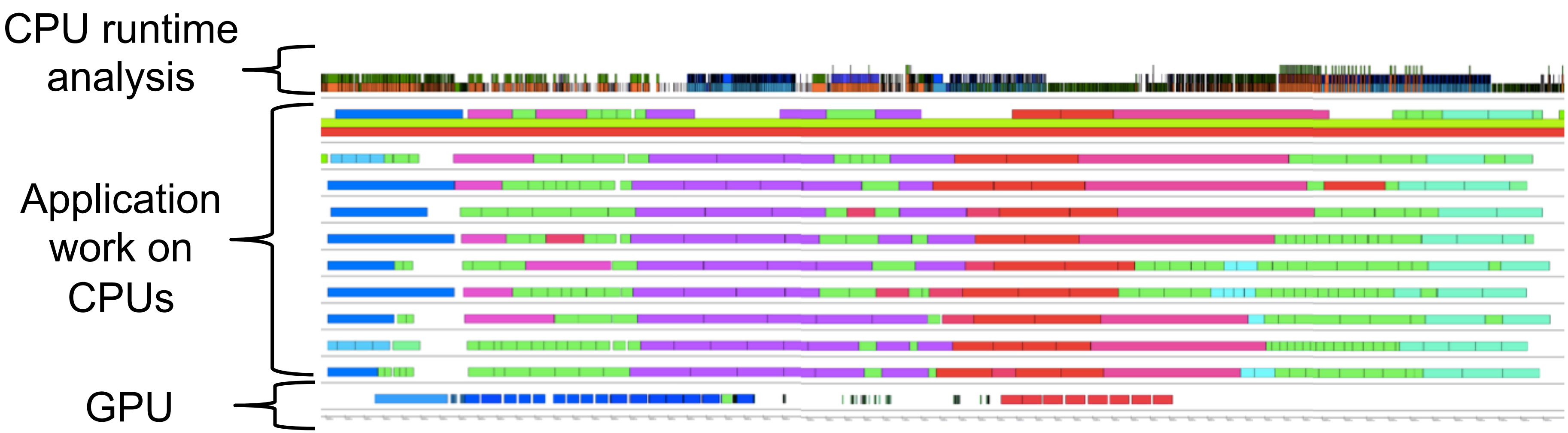
S3D: Heptane 48^3

Profile from one node



S3D: Heptane 96^3

Profile from one node



Problem: 96^3 points per GPU did not fit on the GPU.

Solution: Move some tasks to the CPU to reduce memory pressure.

Impact on Portability & Productivity

- Many more ports of Legion-S3D than MPI-S3D

- Titan
- Summit
- Piz Daint
- Lassen
- Cori
- Perlmutter
- Frontier

- Many more variations of Legion-S3D
  - Different boundary conditions
  - Different reactions

- Example: Simulation of PRF with 116 chemical species
  - The most complex such simulation ever done

Comparison with MPI

- Legion
- Sequential semantics
- Asynchronous by default
- Strong data model
  - System understands the partitioning of data
- Late binding of performance decisions
- Downside: Higher runtime overhead

- MPI
- Explicit parallel programming
- Synchronous by default
- Bag-of-bits data model
- Many performance decisions baked into the code
- Upside: Minimal runtime overhead

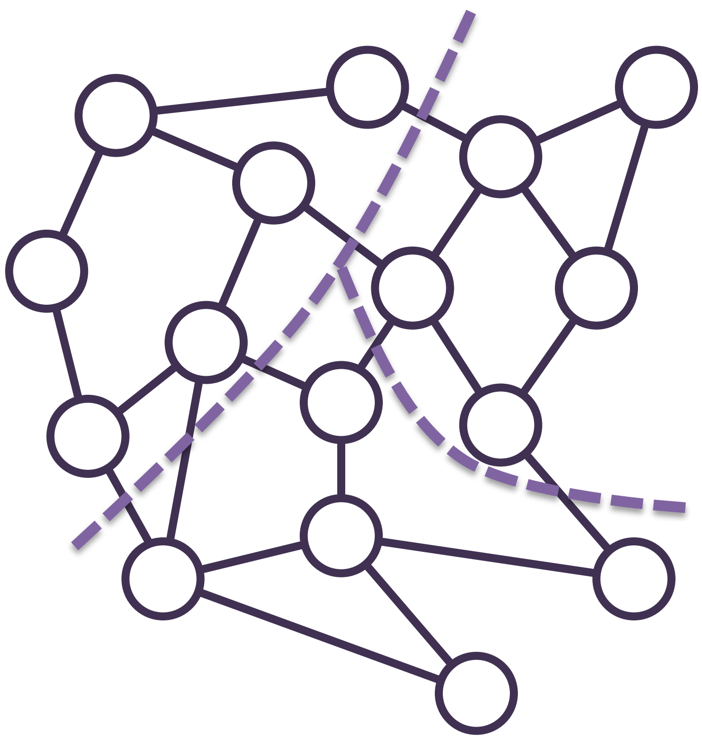
Data in Legion

- Data partitioning
- Partitioning primitives
- Examples

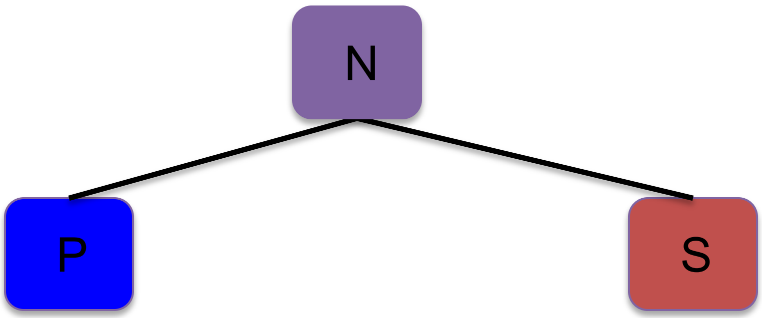
Partitioning

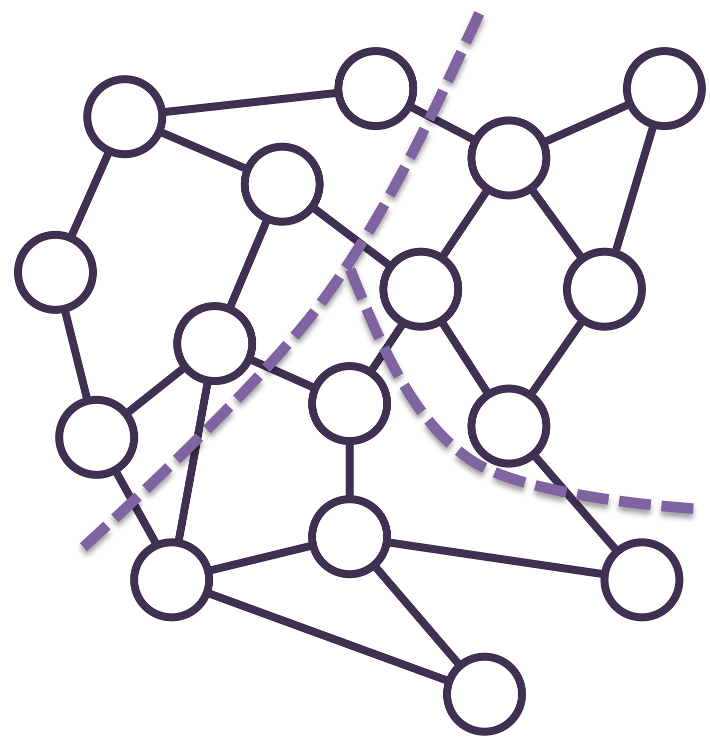
- Partitioning data is a distinctive feature of distributed computing
  - Or whenever there are multiple, distinct memories
- How should data be partitioned?

Partitioning

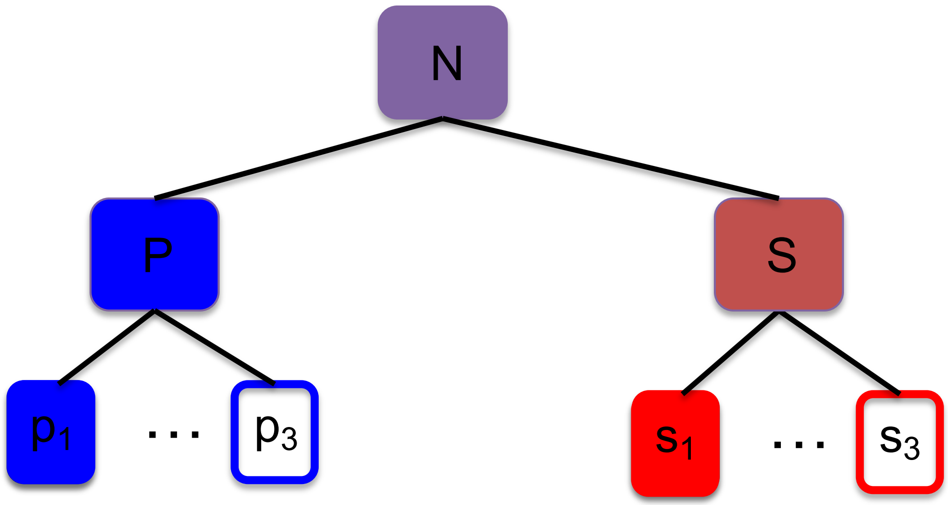


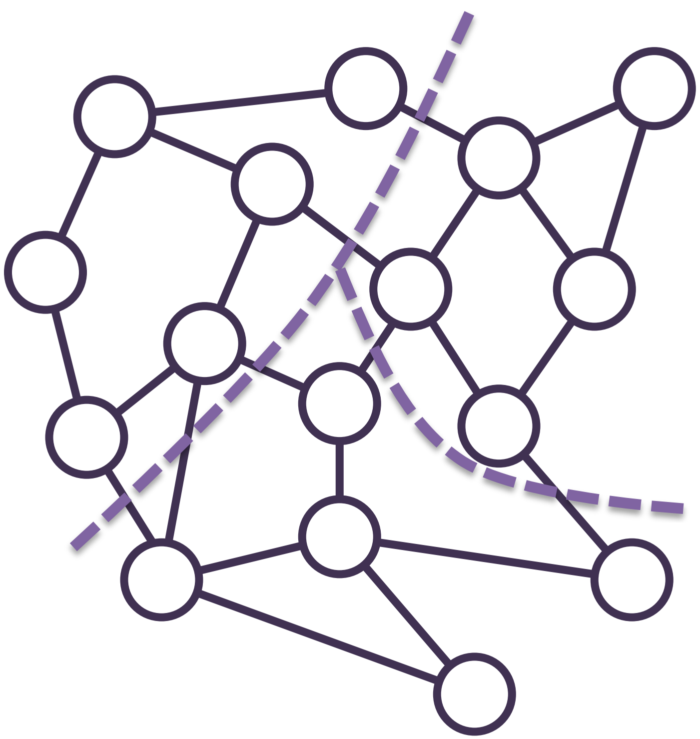
Partitioning



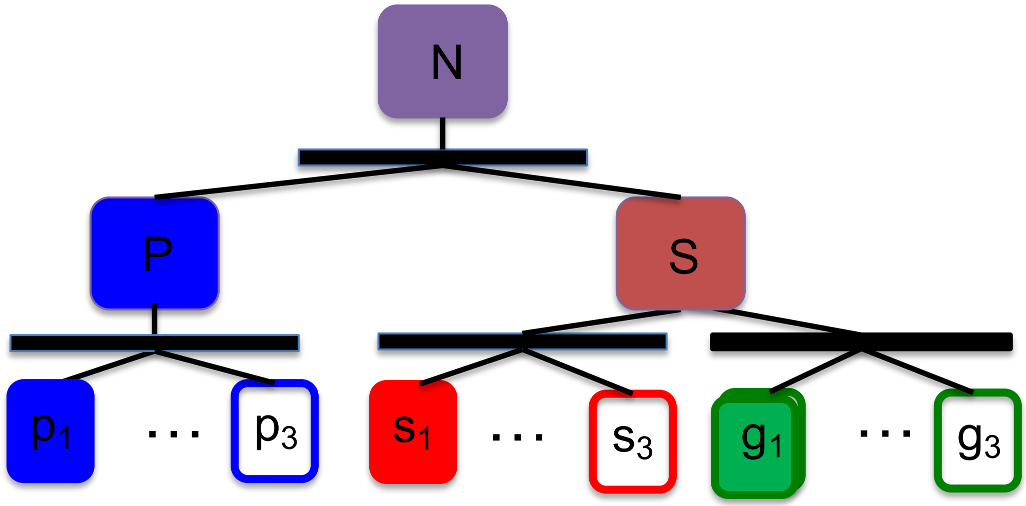


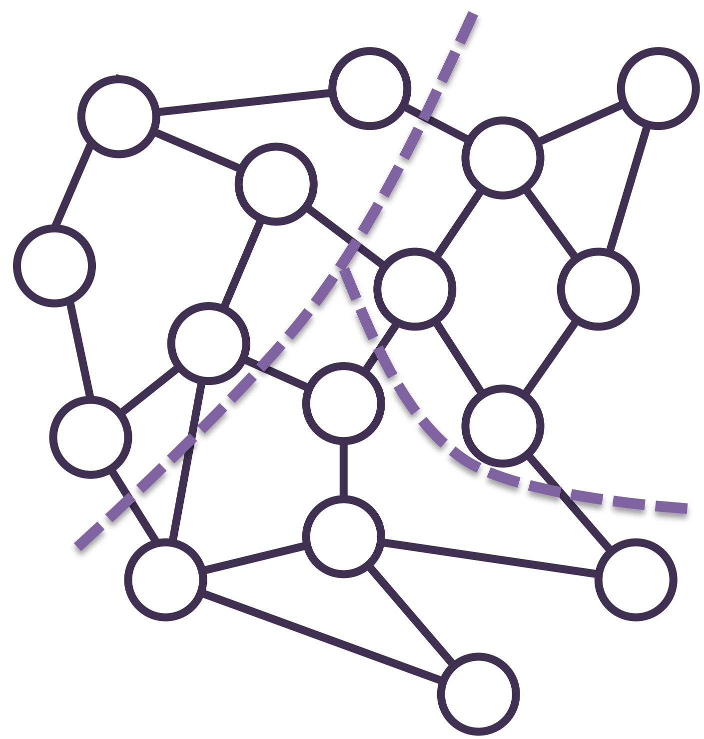
Hierarchical Partitioning





Multiple Partitions





Legion Example

```
task distribute_charge(rpn, rsn, rgn : region(node),
                       rw : region(wi   )
where
  reads
  reduc
{
```

Tasks are the unit of parallel execution.

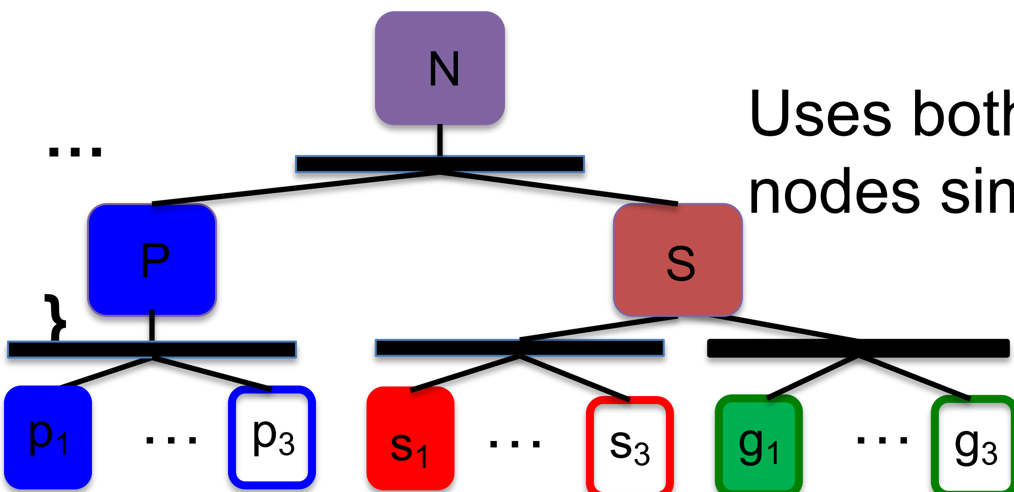
Regions are n-dimensional tables (tensors) with typed columns (fields).

Privileges declare how a task will use its region arguments.

Legion Example

```
task distribute_charge(rpn, rsn, rgn : region(node),
                                     rw : region(wire))
where
  reads(rw.{in_ptr, out_ptr, current})
  reduces +(rpn.charge, rsn.charge, rgn.charge)
{
...
}
```

Uses both views of the shared nodes simultaneously.



Observation: Compositionality

Multiple partitions of the same data are needed for scalable software composition

- **Consider two libraries**
  - **Written independently**
  - **Using different partitioning strategies**
  - **How can they be composed?**

- **Examples**
  - **A simulation, a solver, and a visualization library**
  - **A data analysis pipeline**

Partitioning Operators

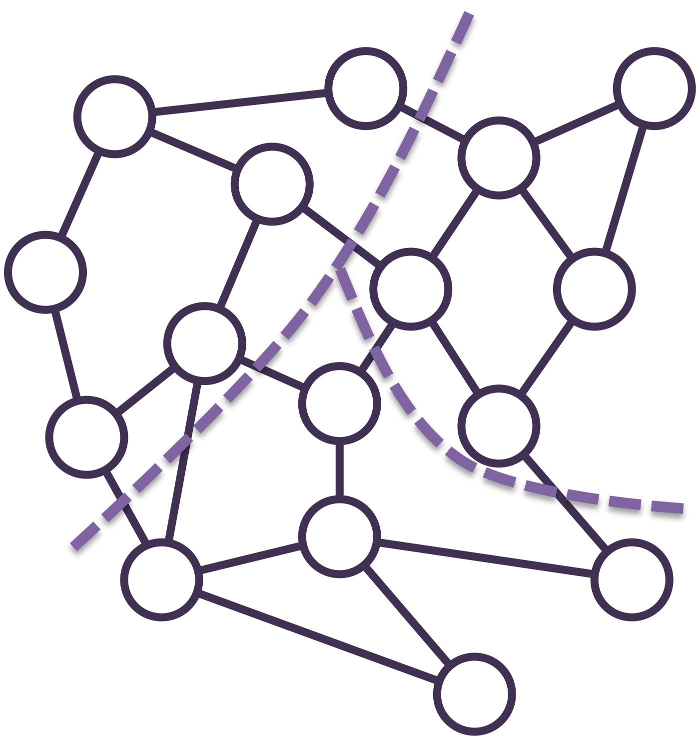
- **Legion has a rich subsystem of partitioning primitives**
- **Each primitive is designed for efficient, scalable parallel implementation**
- **Combinations of primitives express sophisticated partitioning strategies**

Partitioning by Field

PartitionByField(nodes, nodes.SorP)

Nodes

Index	Voltage	SorP
1	1.4	
2	2.5	
3	0.3	
4	6.2	
5	1.4	
6	0.0	
…	…	

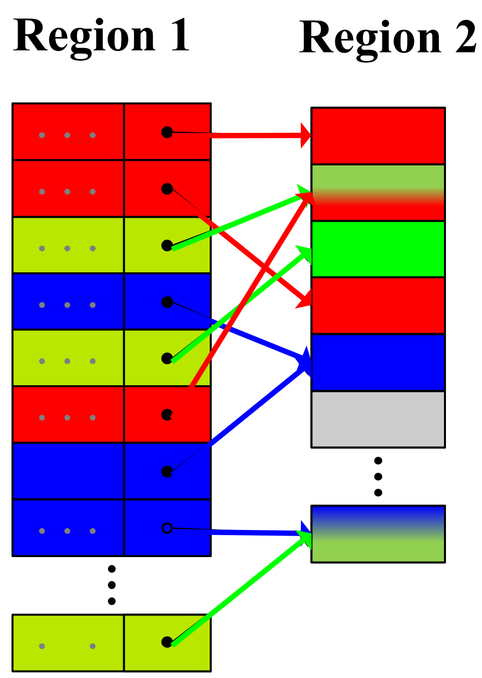


Independent Partitions

- Partitioning by field is an *independent partition*
  - A partitioning that depends on no other partitions
  - Another example: PartitionEqual(R,5)
- Legion also has *dependent partitioning* primitives
  - Compute new partitions from existing partitions
  - Allows regions to be co-partitioned easily

Partition By Image

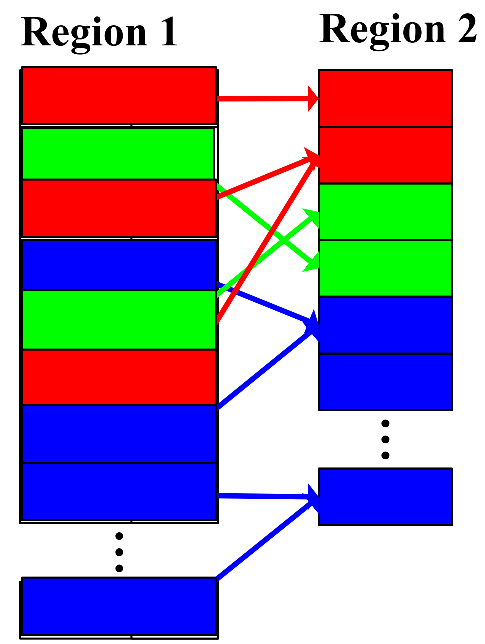
- Treat a pointer field as a function
- Construct compatible partition of destination region
  - Some elements of destination may be in more than one subregion
  - Or in no subregion



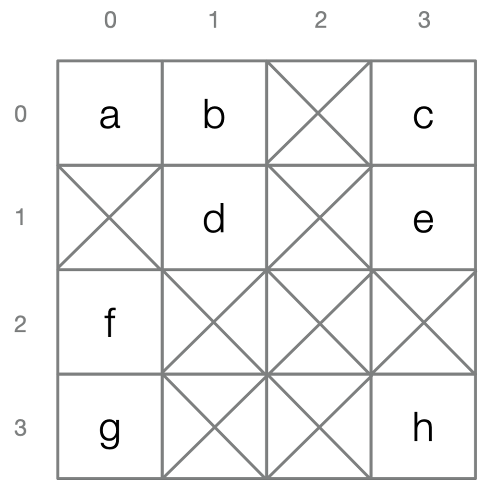
Partition By PreImage

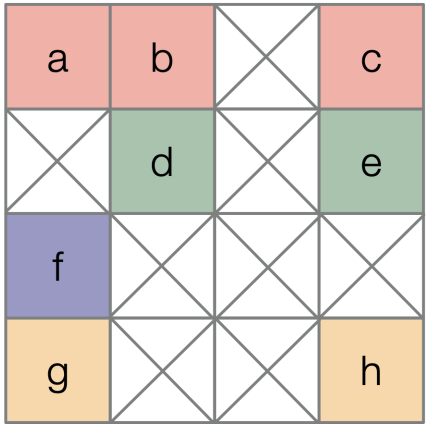
- Again treat a pointer field as a function

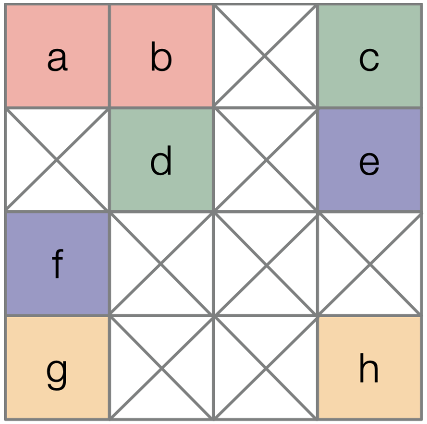
- Construct a compatible partition of the source region



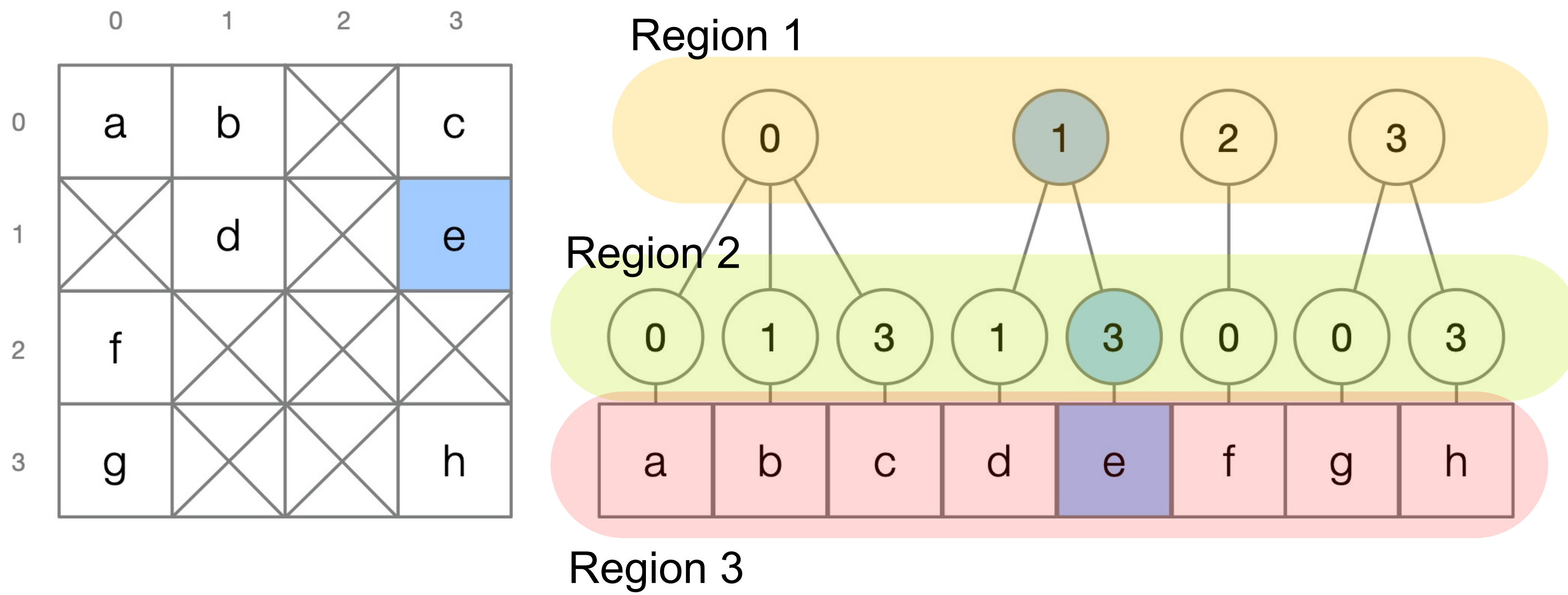
Sparse Matrix Representations



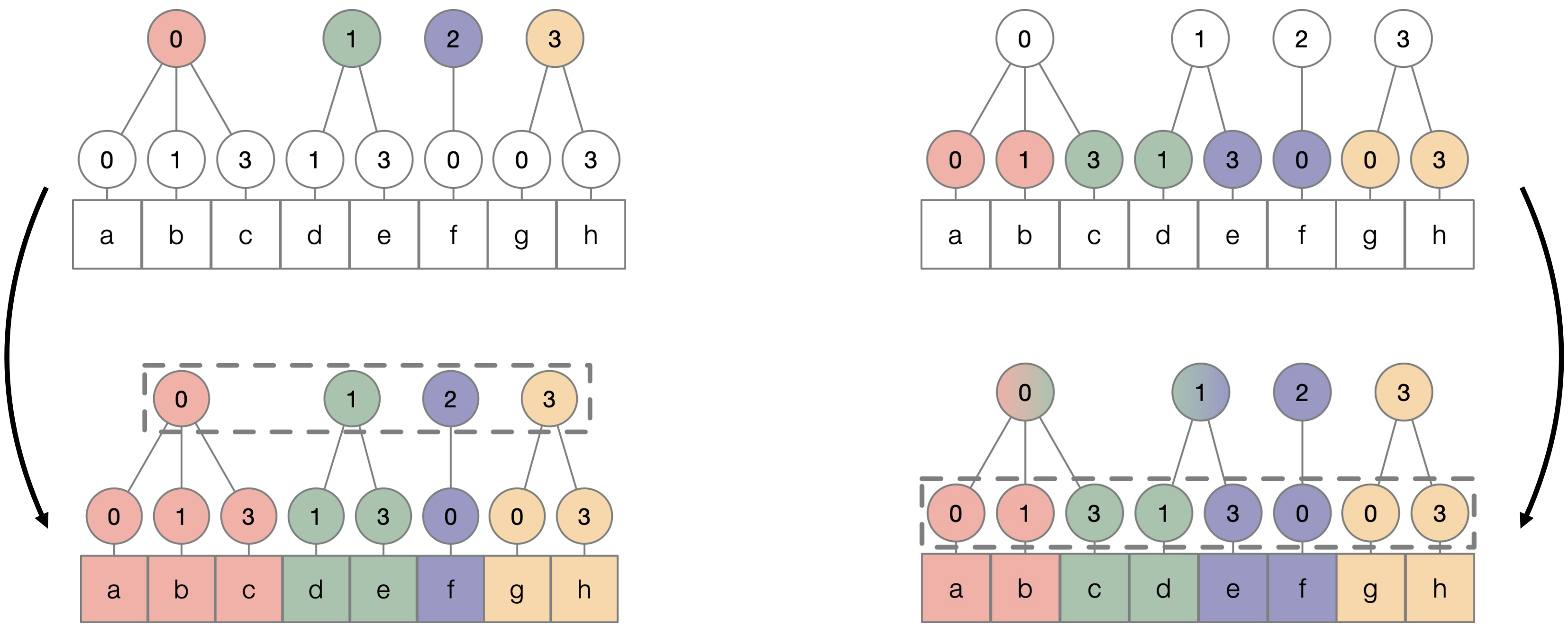




Coordinate Trees

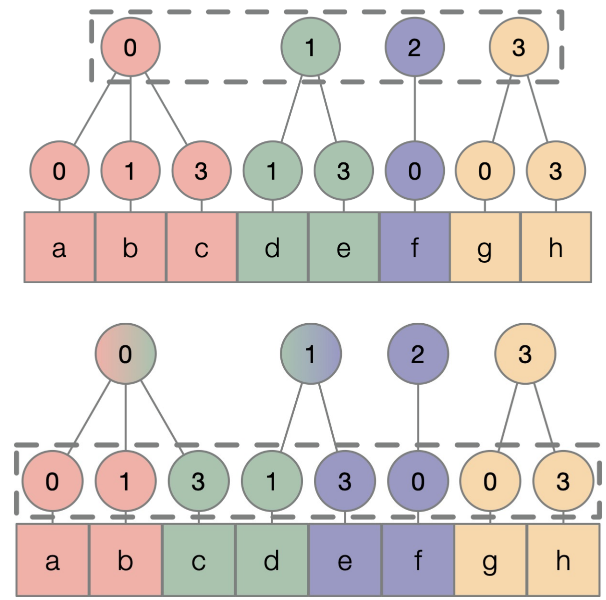


Images and Preimages



Sparse Matrix Partitioning Level-by-Level

- **Partition one level first**
- **Use images and preimages to compatibly partition the other levels**

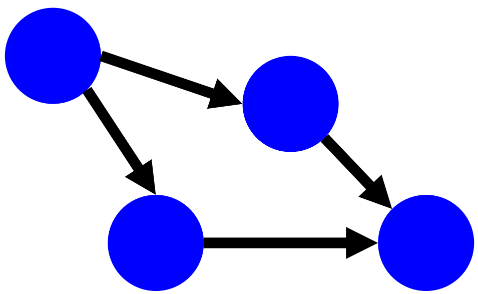


Task-Based Libraries

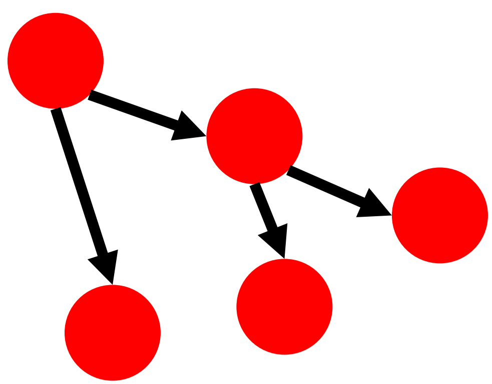
- Task graphs naturally compose
  - Combining two or more task graphs is a task graph
- Late binding of decisions makes interfaces flexible
  - Libraries can be parameterized in ways that are impossible in other approaches
- And we can automate the search for the best partitioning and mapping
  - For a specific machine and workload

Task-Based Libraries

Task1(Args)



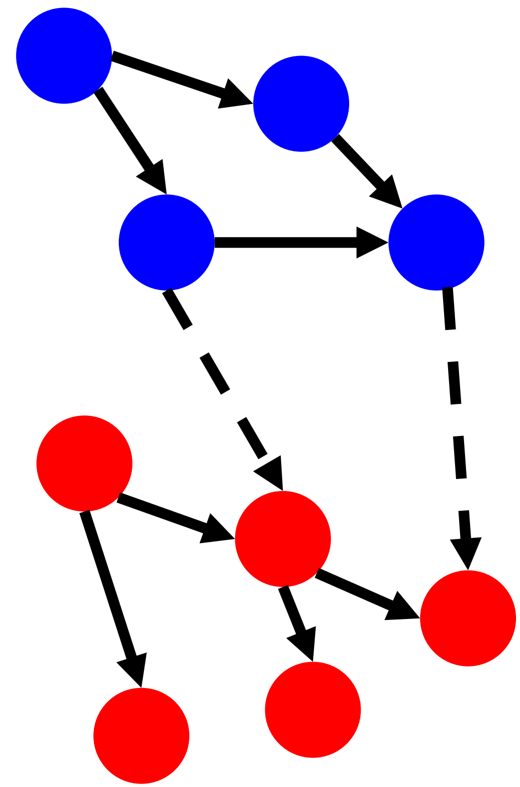
Task2(Args')



Composed:

Task1(Args)

Task2(Args')



DISTAL & SpDISTAL

- DISTAL is a Legion system for dense tensor algebra
- SpDISTAL is a variant for sparse tensor algebra

$$A(i,j) = B(i,k) * C(k,j)$$

$$A(i,j) = \sum_{k} B(i,k) * C(k,j)$$

- DISTAL is a DSL for tensor algebra
  - Given an expression *e* in tensor algebra, generate a task-based library to compute *e*
  - Integrated with a compiler to generate tuned kernels

Distributed Dense Matrix Multiply

Describe an n-dimensional target machine

Schedule describes how kernel interacts with the distributed data

Cannon's Algorithm (1969)

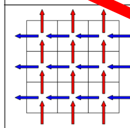
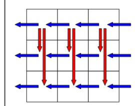
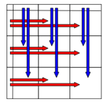
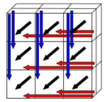
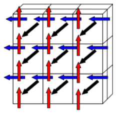
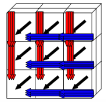
PUMMA (1994)

SUMMA (1995)

Johnson's Algorithm (1995)

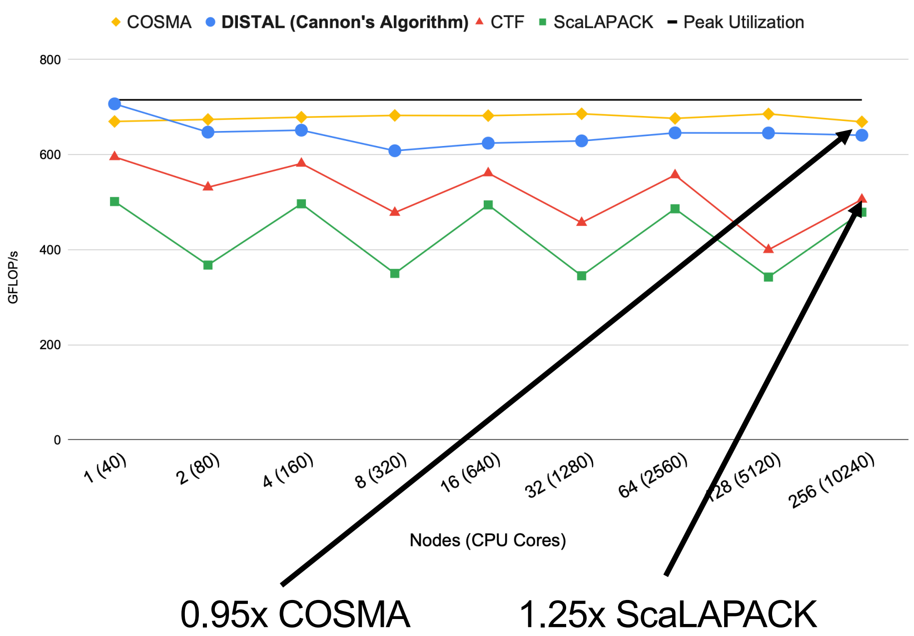
Solomonik's Algorithm (2011)

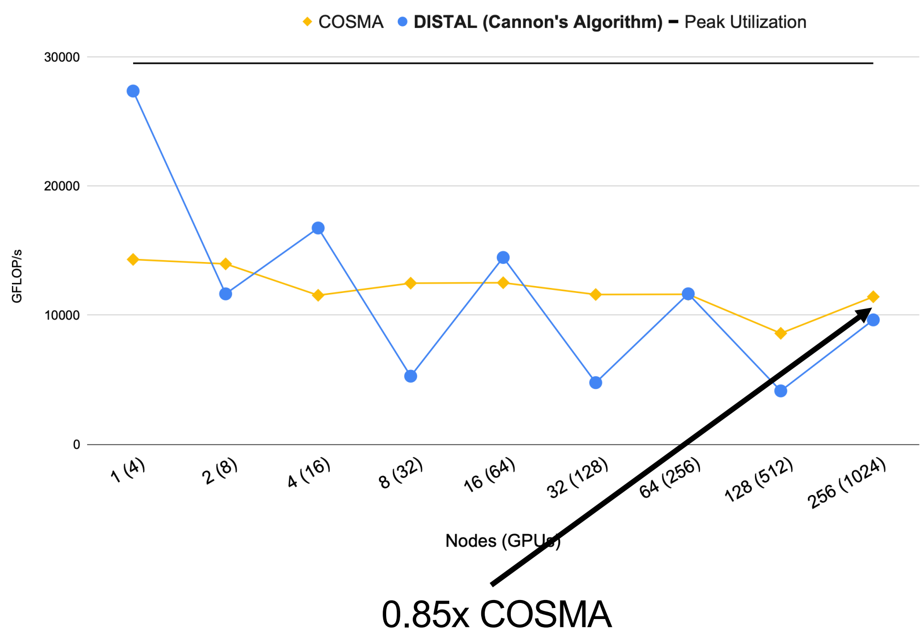
COSMA (2019)

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)
	$\mathcal{M}(gx, gy)$	$A_{ij} \mapsto \mathcal{M}$ $B_{ij} \mapsto \mathcal{M}$ $C_{ij} \mapsto \mathcal{M}$	.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)
	$\mathcal{M}(gx, gy)$	$A_{ij} \mapsto \mathcal{M}$ $B_{ij} \mapsto \mathcal{M}$ $C_{ij} \mapsto \mathcal{M}$	.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)
	$\mathcal{M}(\sqrt[3]{p}, \sqrt[3]{p}, \sqrt[3]{p})$	$A_{ij} \mapsto_{i} \mathcal{M}$ $B_{ik} \mapsto_{i\,k} \mathcal{M}$ $C_{kj} \mapsto_{(\,k} \mathcal{M}$	.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)
	$\mathcal{M}(\sqrt{\frac{p}{c}}, \sqrt{\frac{p}{c}}, c)$	$A_{ij} \mapsto_{i} \mathcal{M}$ $B_{ij} \mapsto_{i} \mathcal{M}$ $C_{ij} \mapsto_{i} \mathcal{M}$	.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)

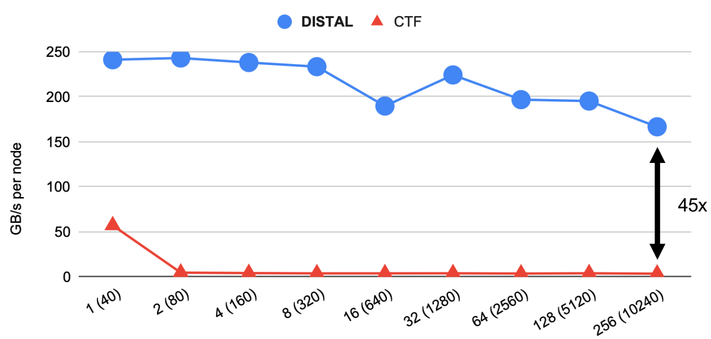
Data partitioning and distribution

Comparison with MM Libraries

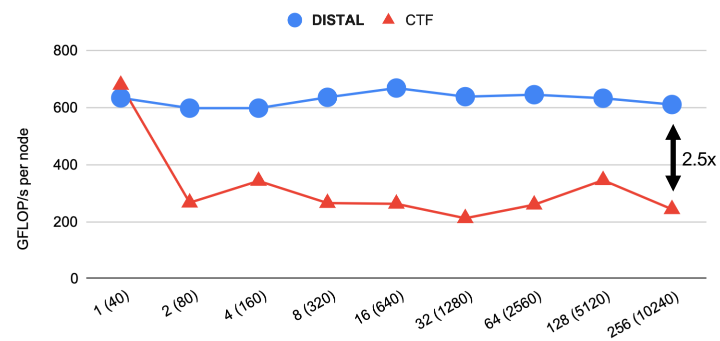




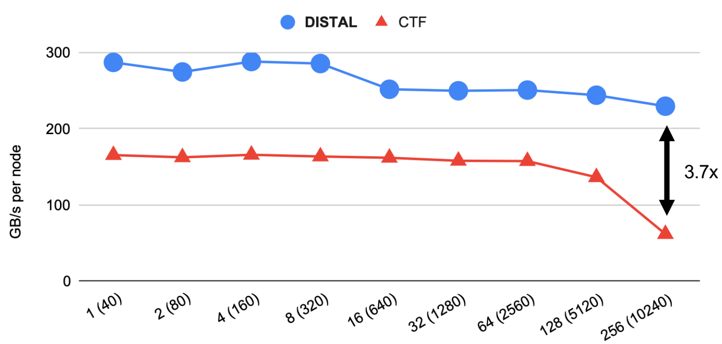
Generalizes to All of Tensor Algebra (CPUs)



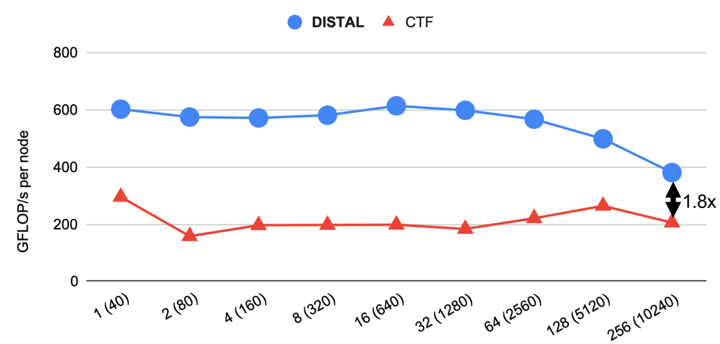
$A(i,j) = B(i,j,k) \cdot c(k)$ Nodes (CPU Cores)



$A(i,j,l) = B(i,j,k) \cdot C(k,l)$ Nodes (CPU Cores)

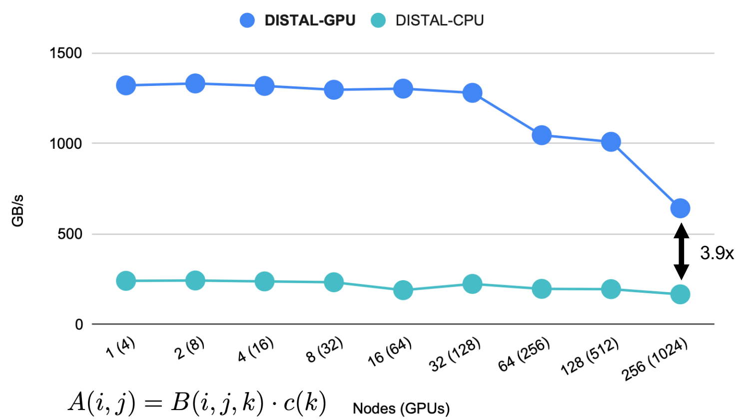


$a = B(i,j,k) \cdot C(i,j,k)$ Nodes (CPU Cores)

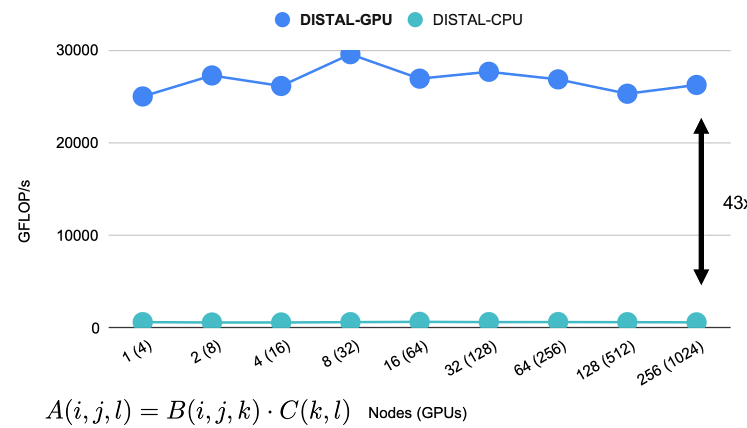


$A(i,l) = B(i,j,k) \cdot C(j,l) \cdot D(k,l)$ Nodes (CPU Cores)

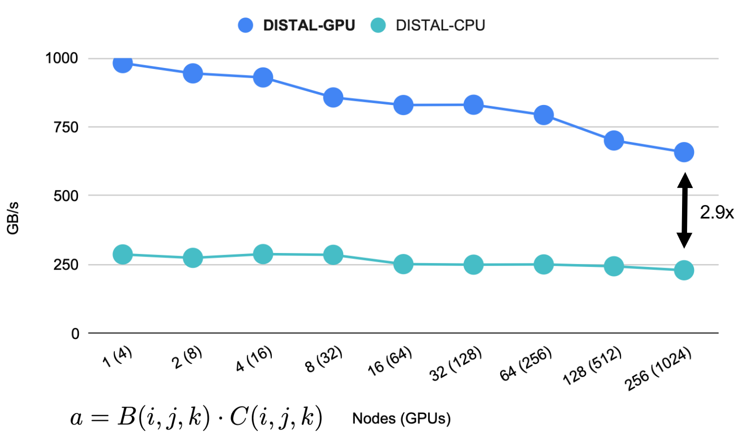
Generalizes to All of Tensor Algebra (GPUs)



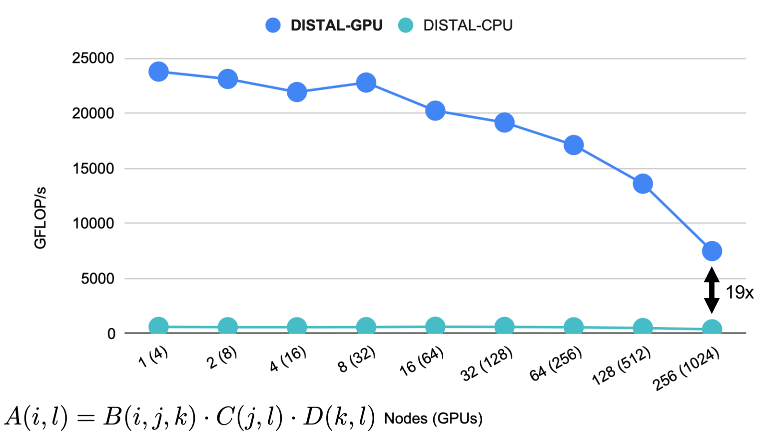
$A(i,j) = B(i,j,k) \cdot c(k)$



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

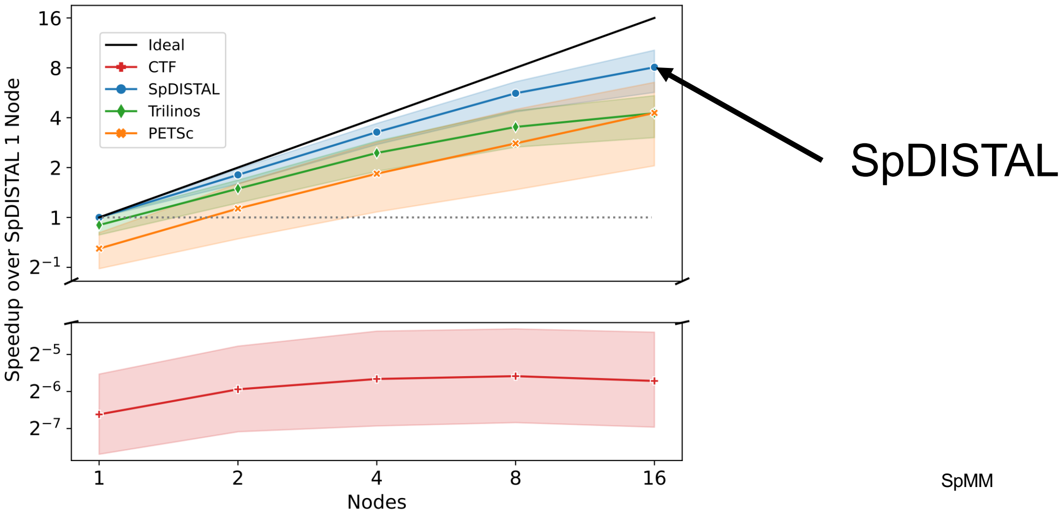


$a = B(i,j,k) \cdot C(i,j,k)$

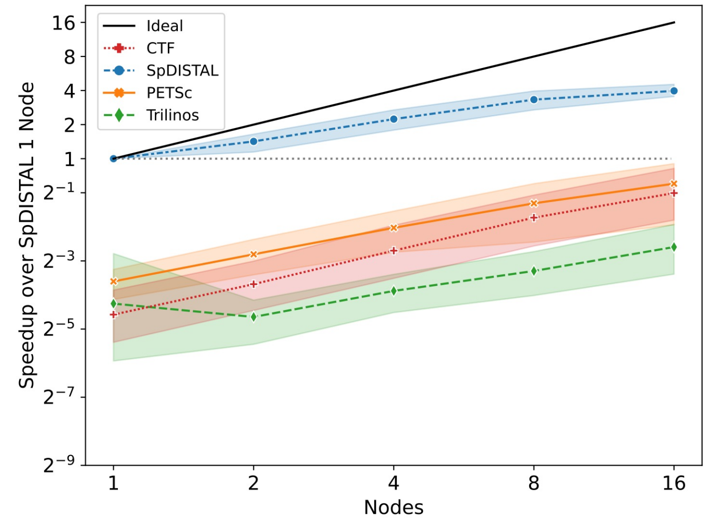


$A(i,l) = B(i,j,k) \cdot C(j,l) \cdot D(k,l)$

And Sparse Tensor Algebra

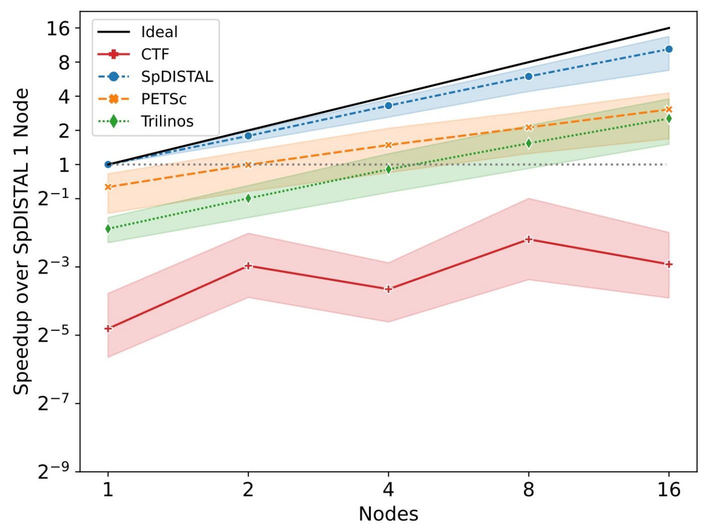


SpMV



SpAdd3

SpMM



FlexFlow: Deep Neural Networks

- FlexFlow is a Legion library for DNN training and inference

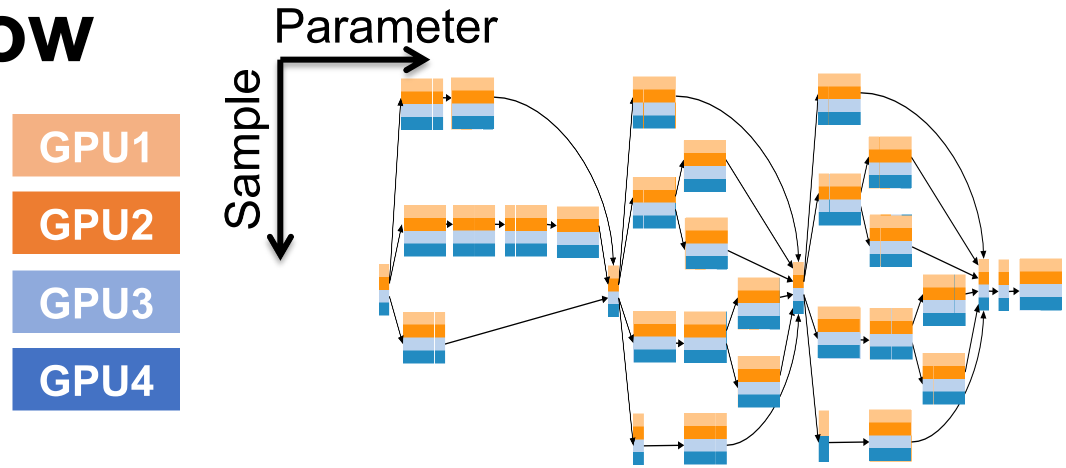
- Idea #1: Exploit Legion's expressive data partitioning to partition tensors in DNN's in ways that Pytorch and TensorFlow do not consider
  - E.g., tensor = [image, height, width, channel]
  - Standard approaches partition the image dimension

- FlexFlow can partition/parallelize data/computations in many more dimensions

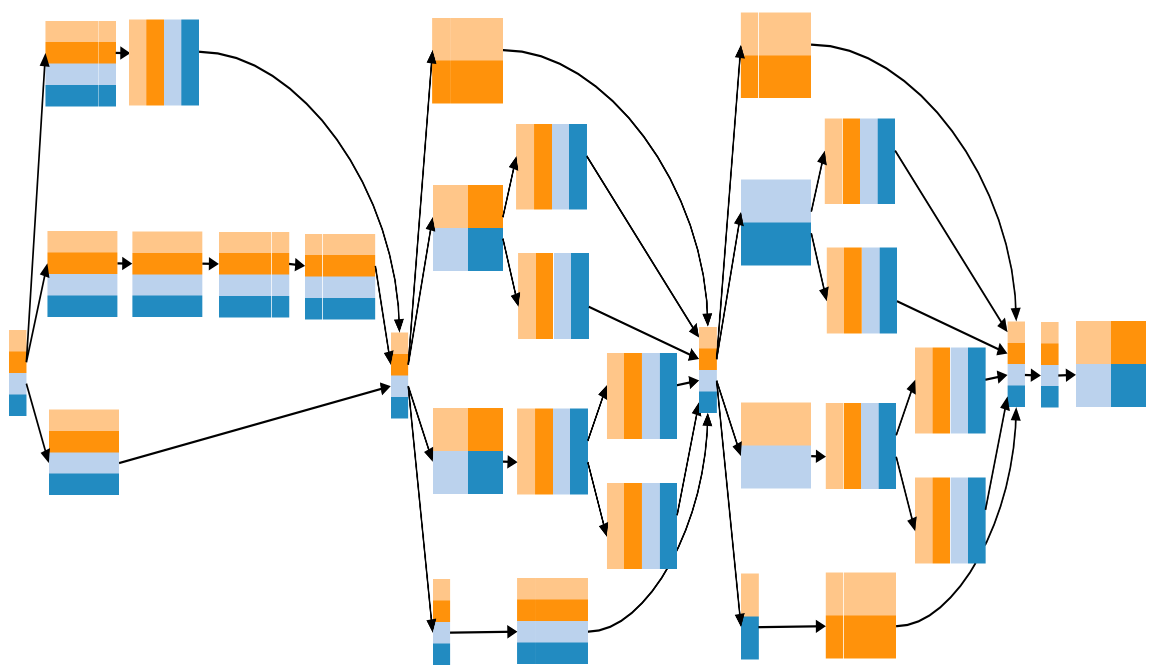
FlexFlow: Deep Neural Networks

- Idea #2: Automate the partitioning process
  - Instead of searching for a good partitioning by hand
- Use the fact that program structure remains the same – only the partitioning of data changes
- And do this for every layer of the network
  - Allow different layers to have different partitioning strategies

FlexFlow

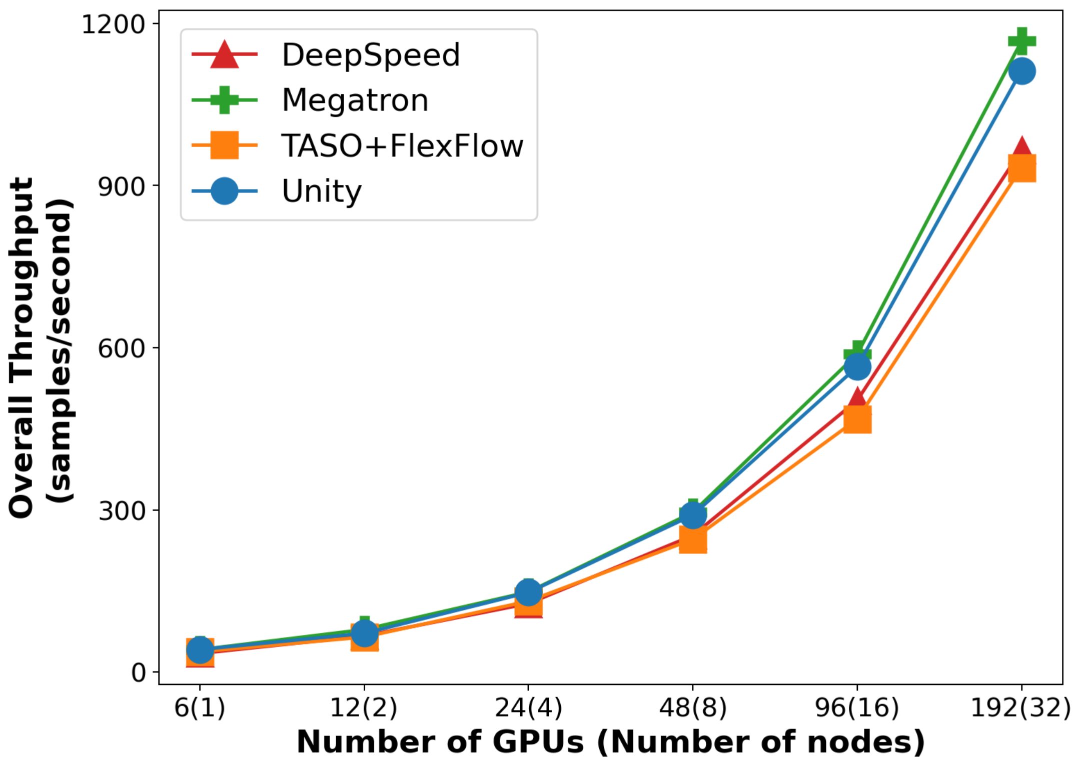


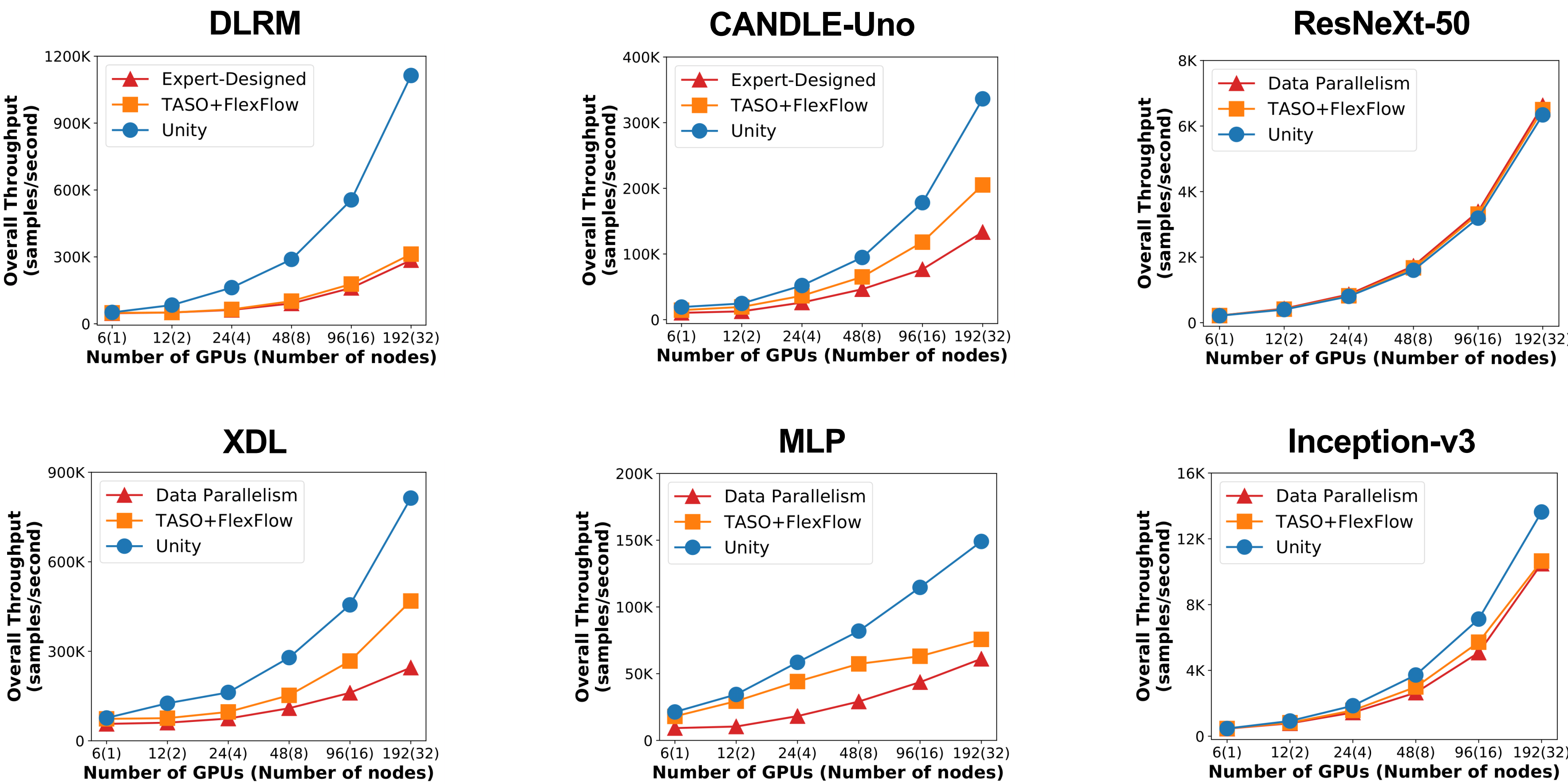
Data parallelism



Results: Bert-Large

Unity is the latest version of FlexFlow …




DLRM
CANDLE-Uno
ResNeXt-50
XDL
MLP
Inception-v3
Overall Throughput (samples/second)
Number of GPUs (Number of nodes)
6(1) 12(2) 24(4) 48(8) 96(16) 192(32)
Expert-Designed
Data Parallelism
TASO+FlexFlow
Unity
0 300K 600K 900K 1200K
0 100K 200K 300K 400K
0 2K 4K 6K 8K
0 300K 600K 900K
0 50K 100K 150K 200K
0 4K 8K 12K 16K

Selected Other Legion Libraries

- CuNumeric (NVIDIA)
  - A open source, drop-in replacement for NumPy
  - See Seshu Yamajala’s talk at 11:30 on Thursday
- LegionSolvers (in progress)
  - Sparse iterative distributed solvers
- Distributed Sparse SciPy (in progress)

Summary

- Task-based programming systems provide a sequential programming model with implicit parallelism
- Late binding of performance decisions has proven key to achieving the best performance
  - Makes it possible to easily explore a large space of configurations
- Strong data model enables data partitioning that is understood by the system

Questions?