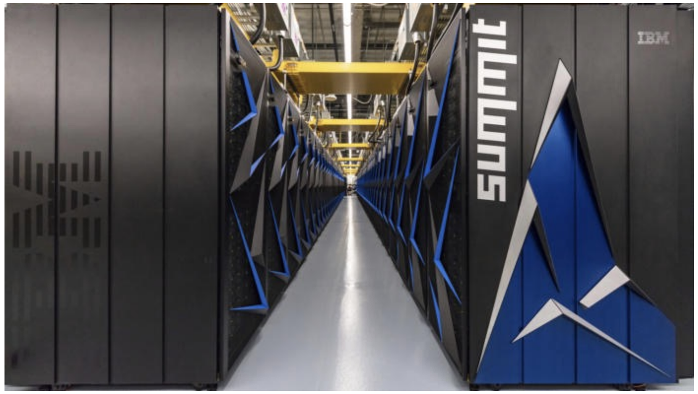
Legion: Programming Distributed, Heterogeneous Architectures

Alex Aiken

Stanford

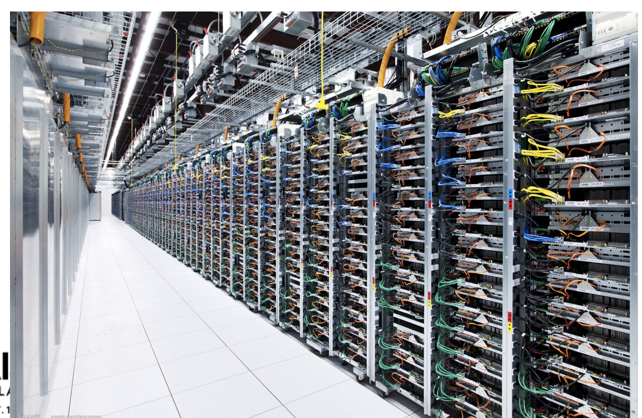
Joint work involving Stanford, NVIDIA, LANL & SLAC

Modern Supercomputers



- Heterogeneity
  - Processor kinds
  - Performance

- Distributed Memory
  - Non-uniform in size & speed



How should we program these machines?

Principle

Data, not compute, matters most.

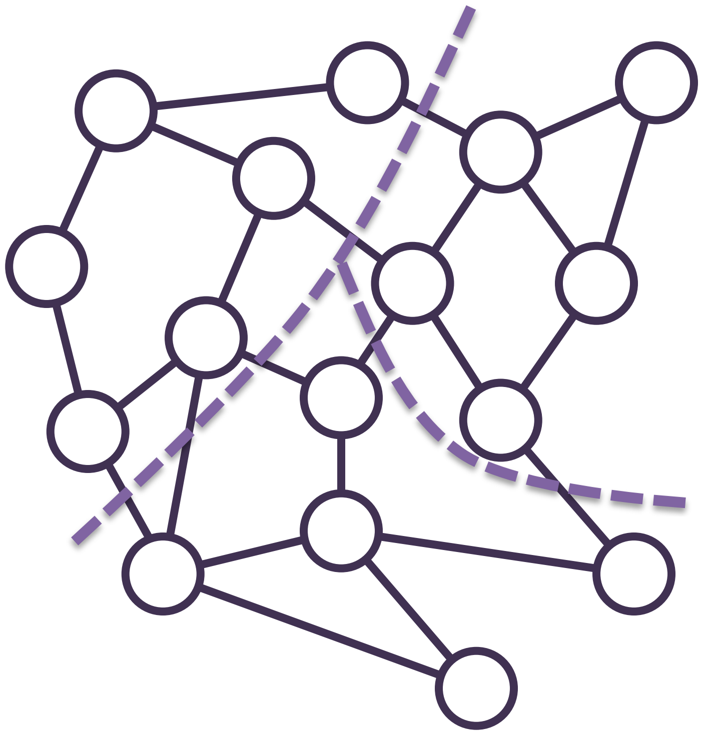
Legion Programming Model Highlights

- Data partitioning
- Partitioning primitives
- Mapping interface
- Control replication

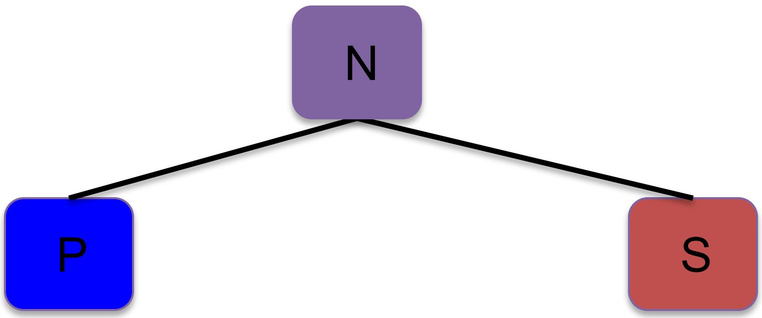
Partitioning

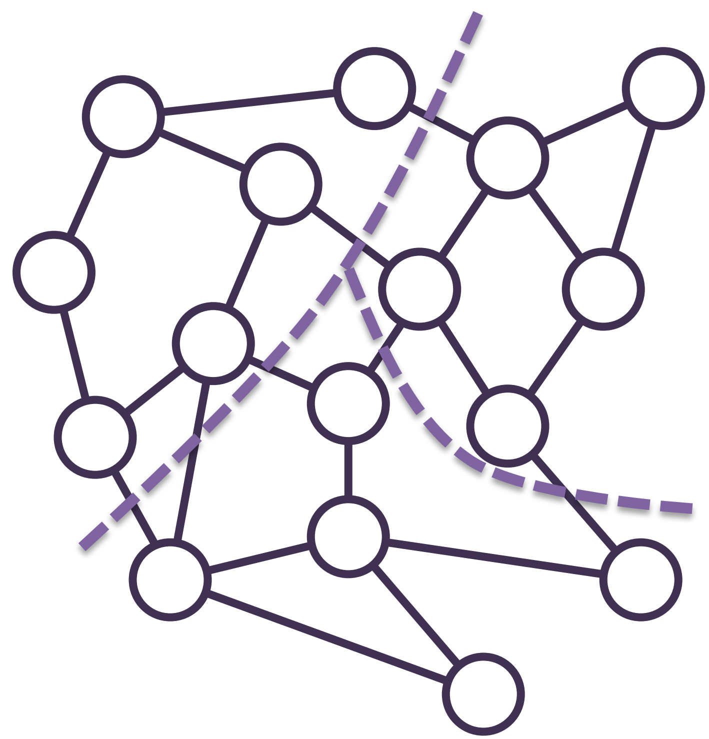
- **Partitioning data is necessary for parallelism**
- **How should data be partitioned?**

Partitioning

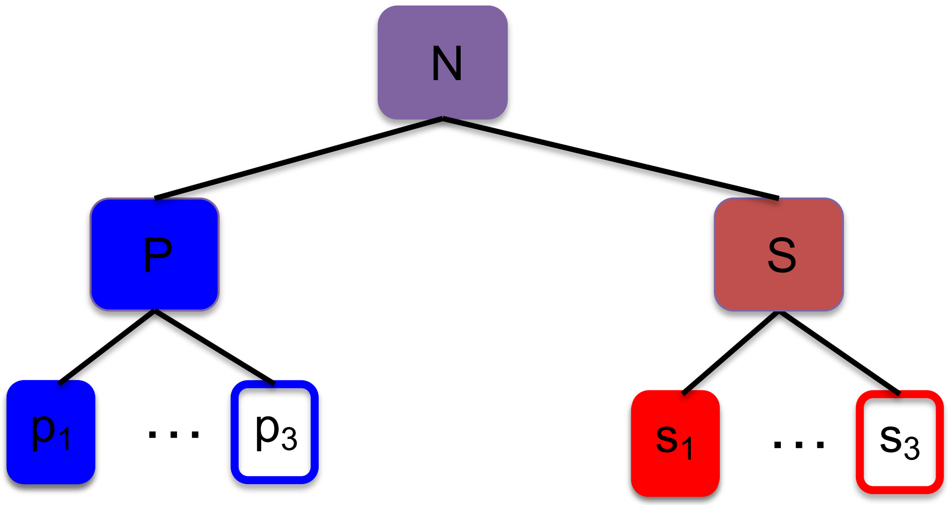


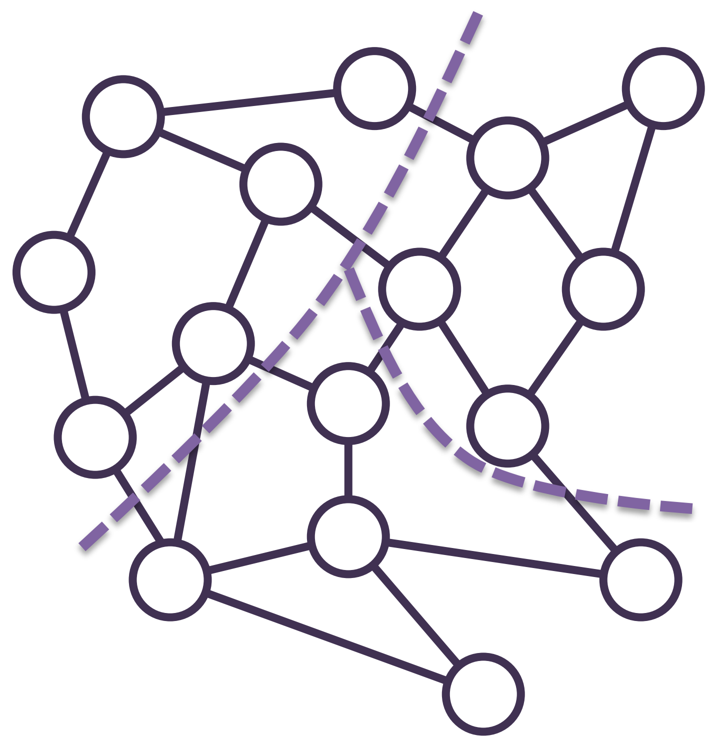
Partitioning



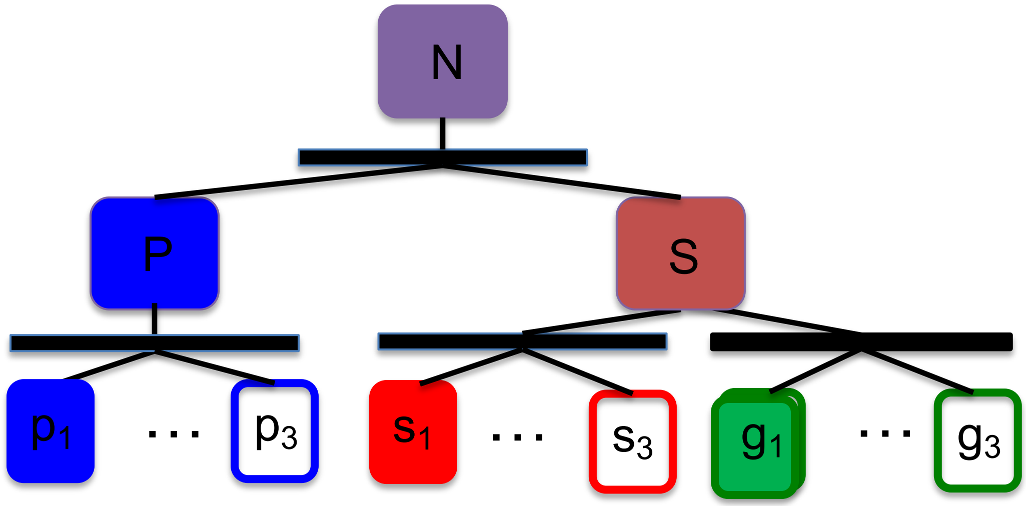


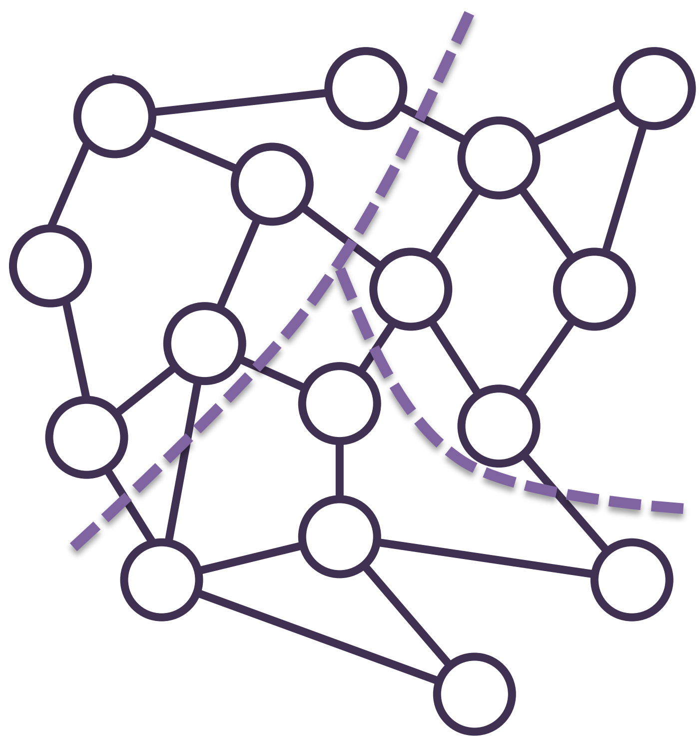
Hierarchical Partitioning





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.

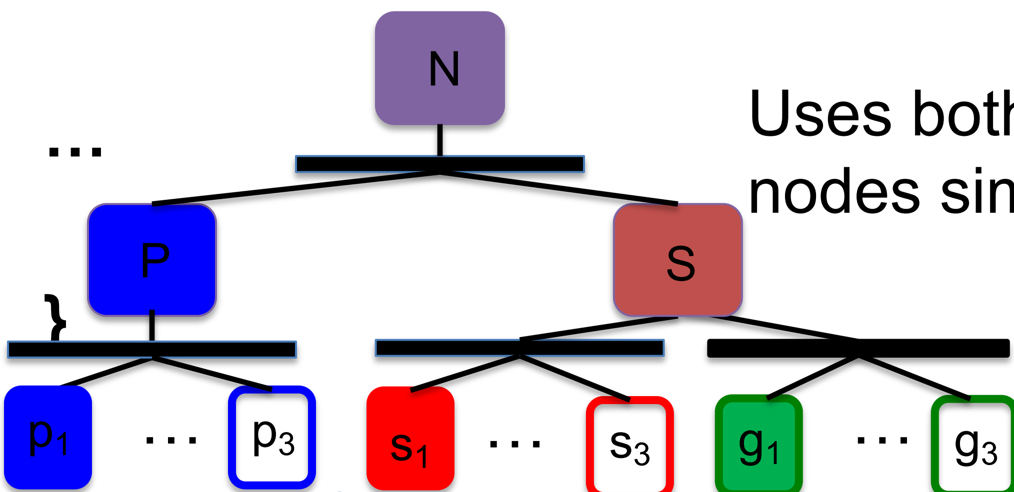
Regions are n-dimensional tables (tensors) with typed columns (fields).

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.



Lesson 1: Compositionality

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

- **Programs use multiple partitions of the same data**
- **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

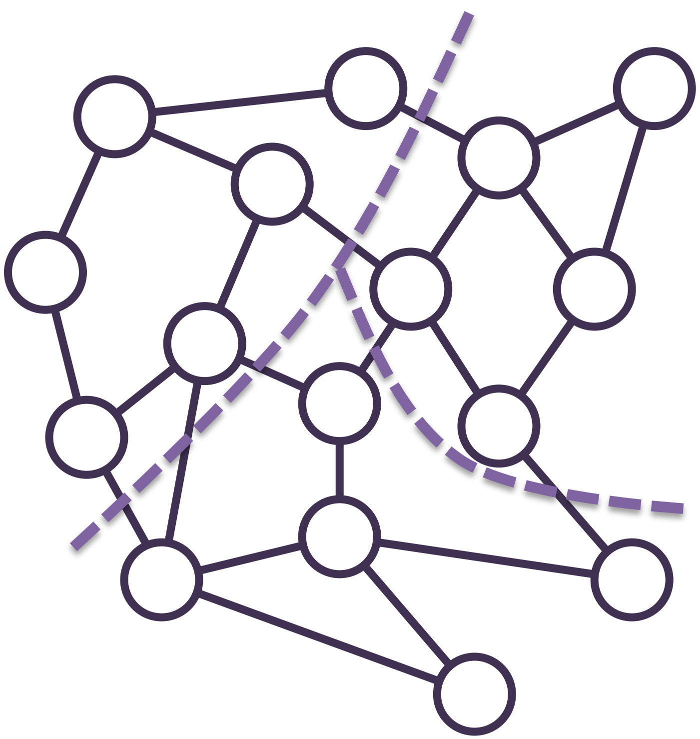
- Legion has a rich subsystem of partitioning primitives
- Each primitive is designed for efficient, scalable parallel implementation
- Combinations of primitives express sophisticated partitioning strategies

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

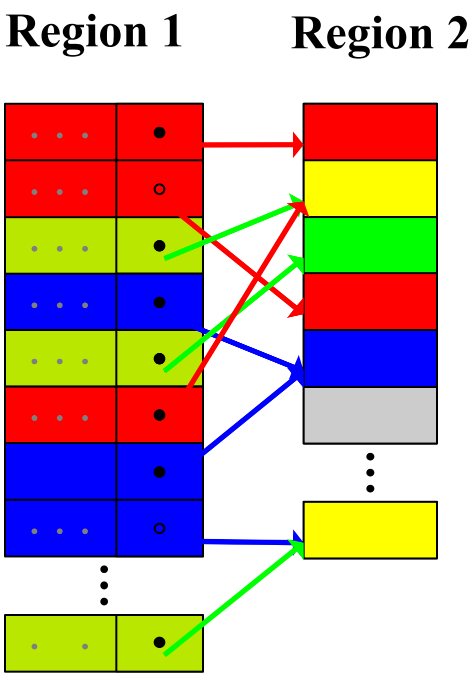
- Partitioning by field is an *independent partition*
  - A partitioning that depends on no other partitions
  - Another example: PartitionEqual(Region,5)

- Legion also has *dependent partitioning* primitives
  - Compute new partitions from existing partitions
  - Allows regions to be co-partitioned easily
  - Set operations (union, intersection, difference of partitions)
  - Image and preimage computations

Partition By Image

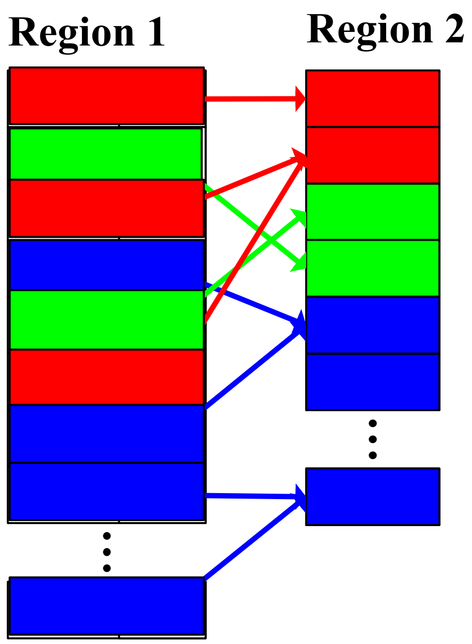
- Treat a pointer field as a function

- Construct compatible partition of destination region



Partition By PreImage

- Again treat a pointer field as a function
- Construct a compatible partition of the source region



Nodes and Edges

Nodes

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

Edges

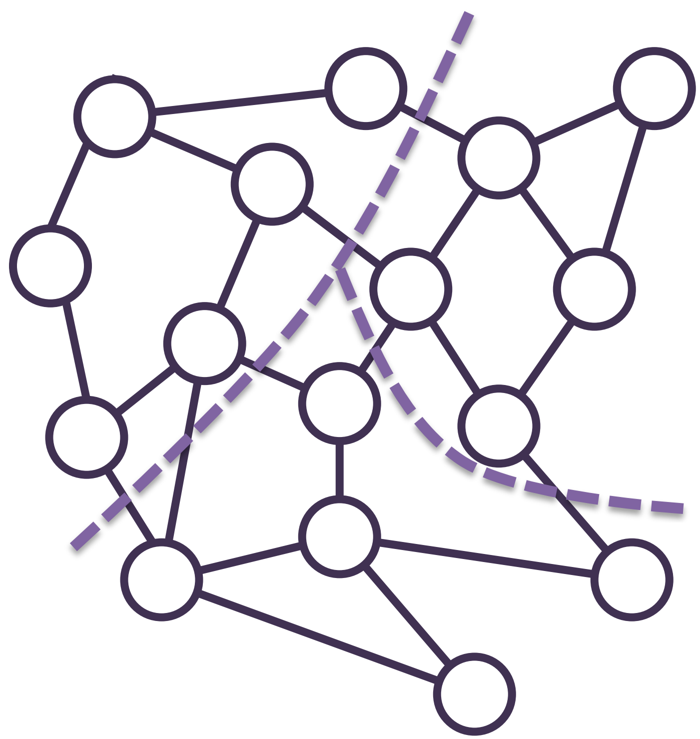
Index	Src	Dst
1		
2		
3		
4		
5		
6		

Dependent Partitioning Example

- Goal: Compute the ghost node partitions

- For each piece
  - Start with the shared nodes of that piece
  - Add adjacent shared nodes
  - Subtract out the shared nodes of that piece

- Computing adjacent nodes of a piece requires an edge partition



Dependent Partitioning Example

NP = PartitionByField(nodes, nodes.SorP)

Priv

Shared

Partitions – arrays of subregions – are first class entities in Legion

PrivatePart = PartitionByField(PrivateNodes, nodes.piece)

SharedPart = PartitionByField(SharedNodes, nodes.piece)

EdgePartSrc = PreImage(edges, SharedPart, edges.src_node)

EdgePartDst = PreImage(edges, SharedPart, edges.dst_node)

EdgePart = EdgePartSrc $\sqcup$ EdgePartDst

SrcNodes = Image(SharedNodes, EdgePart, edges.src_node)

DstNodes = Image(SharedNodes, EdgePart, edges.dst_node)

GhostPart = (SrcNodes $\sqcup$ DstNodes) - SharedPart

Lesson 2: Partitioning Primitives

- Using partitioning primitives is much better than constructing partitions "by hand"
  - More maintainable
  - More performant
  - More scalable
- Requires allowing multiple partitions of data

Legion Tasks

```
task simulate_circuit(Region[Node] N, Region[Wires] W) :
{
    …
    calc_currents( p0 , s0 , g0 );
    calc_currents( p1 , s1 , g1 );
    distribute_charge( p0 , s0 , g0 );
    distribute_charge( p1 , s1 , g1 );
    …
}

task calc_currents(…) :

task distribute_charge(…) :
```

subtasks

Execution Model

```
task simulate_circuit(Region[Node] N, Region[Wires] W) :
{
    …
    calc_currents( p0 , s0 , g0 );
    calc_currents( p1 , s1 , g1 );
    distribute_charge( p0 , s0 , g0 );
    distribute_charge( p1 , s1 , g1 );
    …
}
```

Tasks are issued in program order.

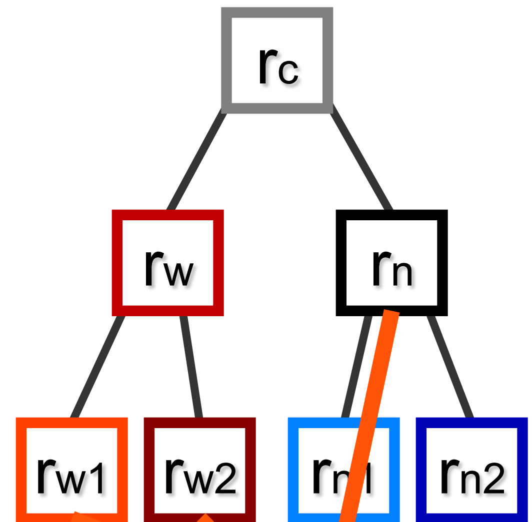
Execution Model

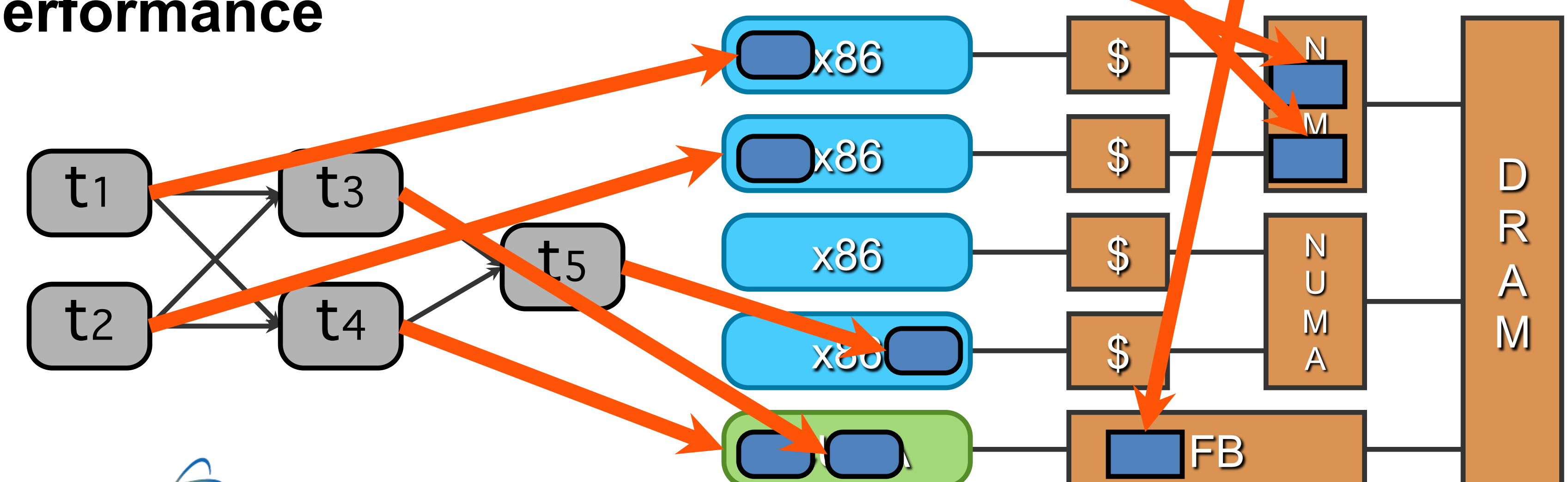
```
task simulate_circuit(Region[Node] N, Region[Wires] W) :
{
    …
    calc_currents( p0 , s0 , g0 );
    calc_currents( p1 , s1 , g1 );
    distribute_charge( p0 , s0 , g0 );
    distribute_charge( p1 , s1 , g1 );
    …
}
```

Tasks without dependences may execute in parallel. Dependence analysis is done dynamically.

Mapping Interface

- **Application selects:**
  - **Where tasks run**
  - **Where regions are placed**
- **Mapping computed dynamically**
- **Decouples correctness from performance**





Lesson 3: Mapping

- **Separation of mapping from program helps enormously with portability**

- **But also enables rapid experimentation and autotuning even on a single machine**
  - **E.g., for different size inputs**

- **Experience shows it is difficult to guess the best mapping**
  - **Late binding of mapping saves recoding**

Legion Tasks

```
task simulate_circuit(Region[Node] N, Region[Wires] W) :
{
 while (not done) {
   …
   calc_currents( p0 , s0 , g0 );
   calc_currents( p1 , s1 , g1 );
   distribute_charge( p0 , s0 , g0 );
   distribute_charge( p1 , s1 , g1 );
   …
 }
}
```

Who launches the subtasks?

Two Answers

- Parent task running on one node
  - A centralized controller
  - And a scalability bottleneck
- Parent task *replicated* across multiple nodes
  - N copies of parent task each do 1/Nth of the work
    - Launch 1/Nth of the subtasks
  - Keeps launch overhead constant in weak scaling
  - Replicas must still implement single task semantics
    - Dependences between different replicas must be preserved

Lesson 4: Control Replication

- Task launch overhead of centralized controller grows rapidly with scale
  - Often cannot scale past 16 or 32 nodes

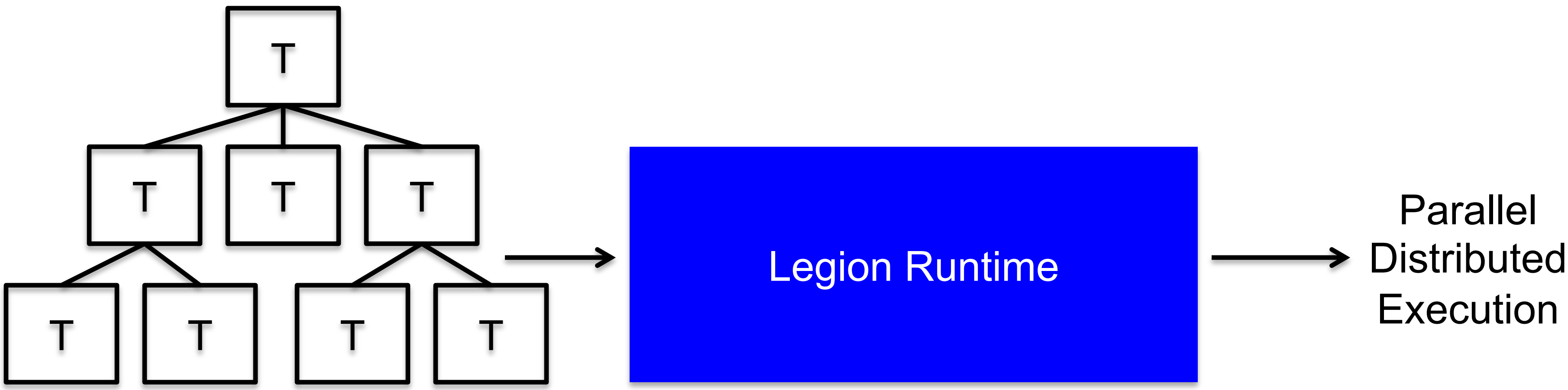
- Control replication
  - Scales to 1,000's of nodes
  - Does not change programming model

Legion Programming Model Summary

- Region-based data model
  - Similar to dataframes, relations, other collections
  - First-class partitioning
  - Allow arbitrary number of views (partitions) of the daa
- Implicit task parallelism
  - Task may have arbitrary subtasks
  - Tasks declare privileges on regions
- Tasks appear to execute in program order
  - Execute in parallel when data dependences permit
- Portability by separating mapping from function

Legion Runtime System

Legion Runtime System



Parent Task

$t_{0:}$: r_5, r_7
t_1: r_0, r_2
t_2: r_1, r_2
t_3: r_4, r_6

Tasks : Regions :: Instructions : Registers

Dep. Analysis	Map	Distribute	Execute	Resolve Spec.	Complete	Commit

A Distributed Hierarchical Out-of-Order Task Processor

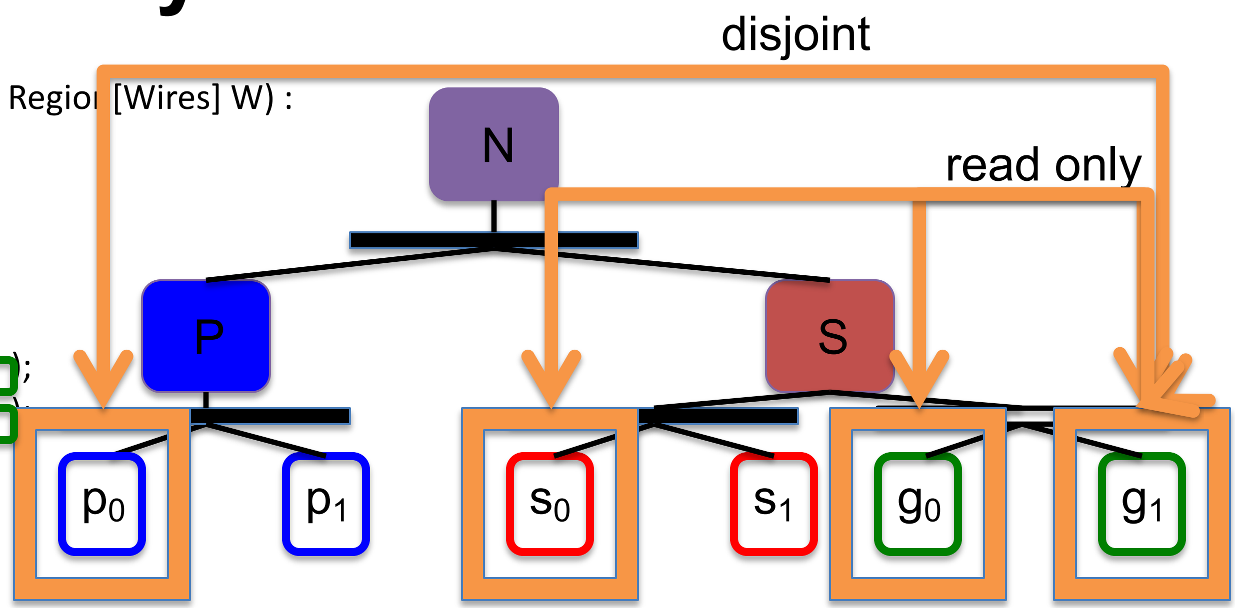
NVIDIA Los Alamos NATIONAL LABORATORY EST. 1943 SLAC

Dependence Analysis

```
task simulate_circuit(Region[Node] N, Region[Wires] W) :
                ReadWrite(N,W)
{
   ...
   calc_currents(piece[0], ...);
   calc_currents(piece[1], ...);
   distribute_charge(piece[0], ...);
   distribute_charge(piece[1], ...);
   ...
}
```

```
task calc_currents(Piece p) :
ReadWrite(p.wires),  Read(p.private, p.shared, p.ghost)
```

```
task distribute_charge(Piece p) :
Read(p.wires), Reduce(p.private, p.shared, p.ghost)
```



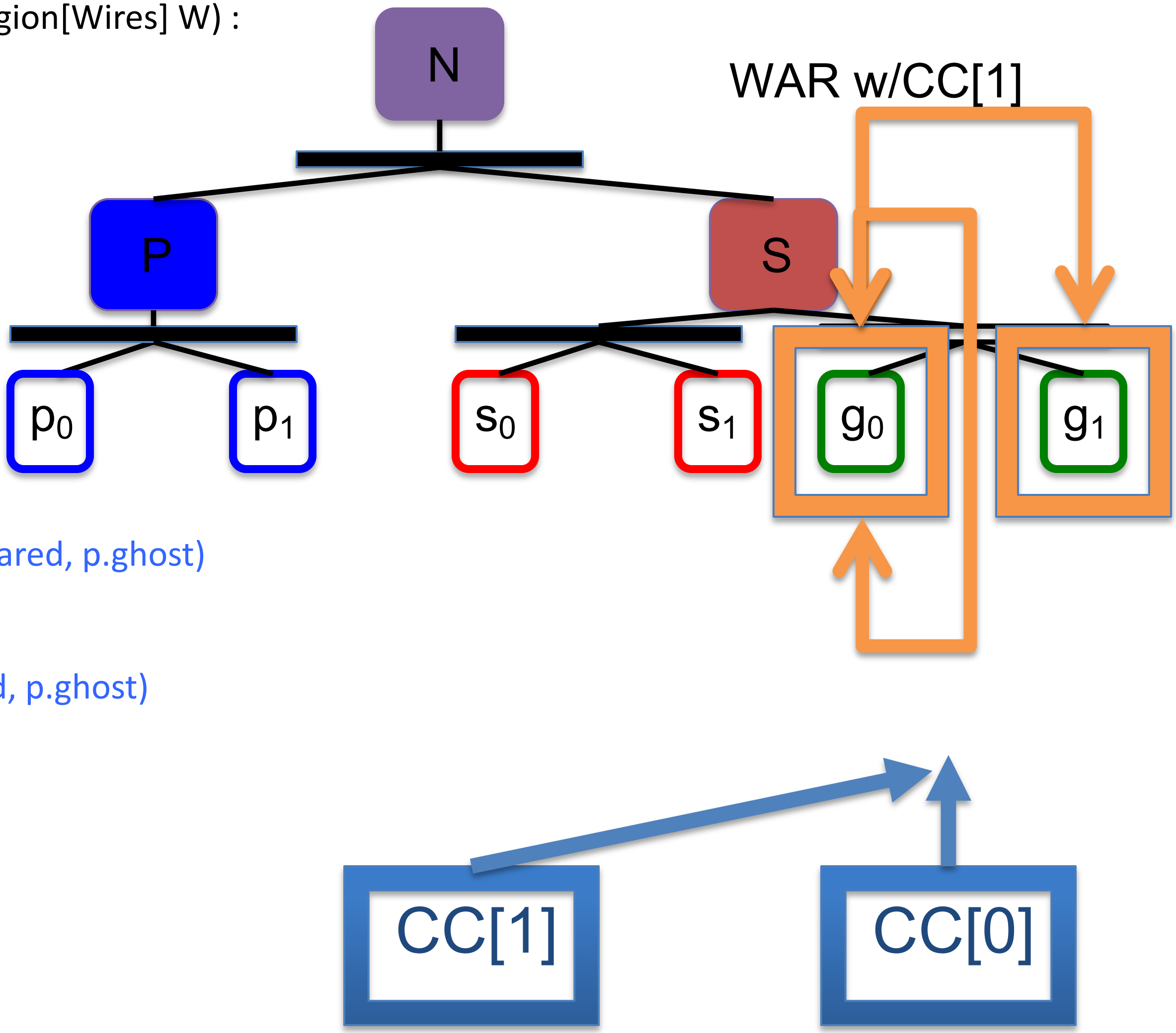
CC[0]

Dependence Analysis

```
task simulate_circuit(Region[Node] N, Region[Wires] W) :
                ReadWrite(N,W)
{
   …
   calc_currents(piece[0], □□□);
   calc_currents(piece[1], □□□);
   distribute_charge(piece[0], □□□);
   distribute_charge(piece[1], □□□);
   …
}
```

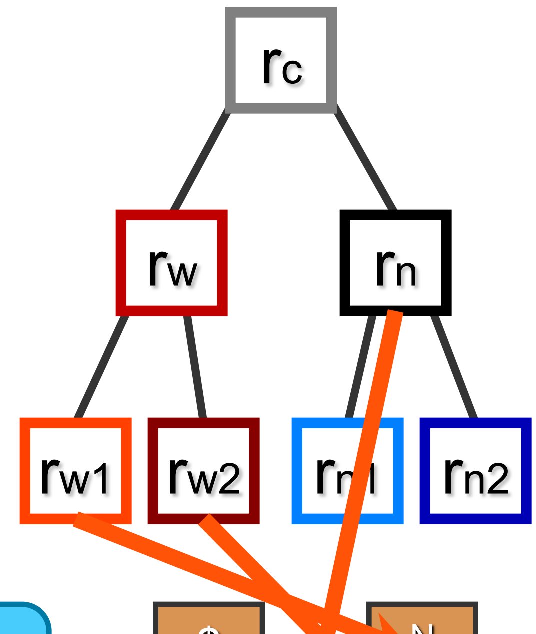
```
task calc_currents(Piece p) :
ReadWrite(p.wires),  Read(p.private, p.shared, p.ghost)
```

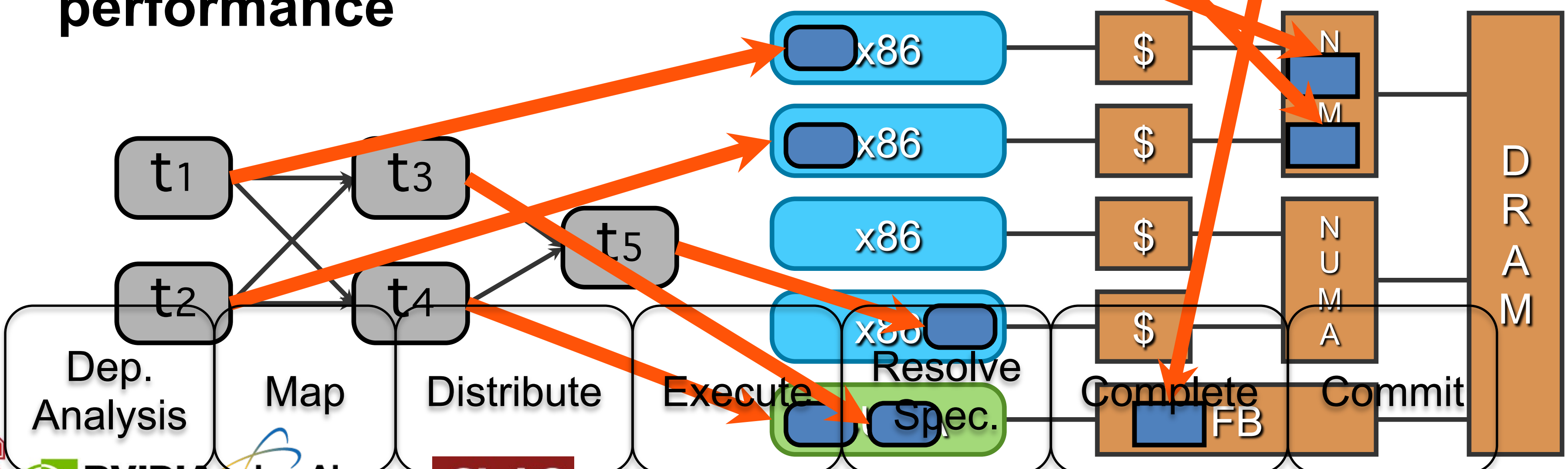
```
task distribute_charge(Piece p) :
Read(p.wires), Reduce(p.private, p.shared, p.ghost)
```



Mapping Interface

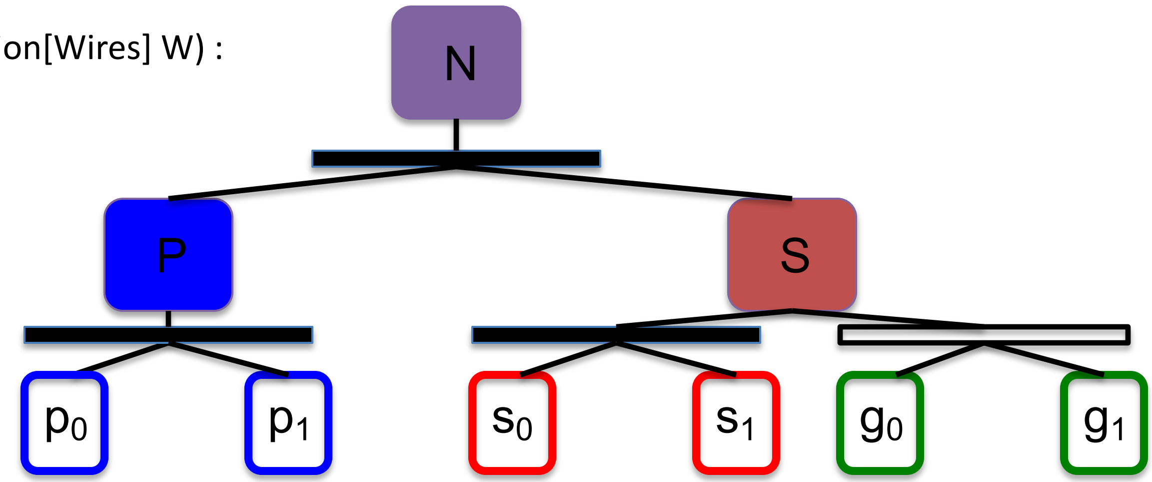
- **Application selects:**
  - **Where tasks run**
  - **Where regions are placed**
- **Mapping computed dynamically**
- **Decouple correctness from performance**



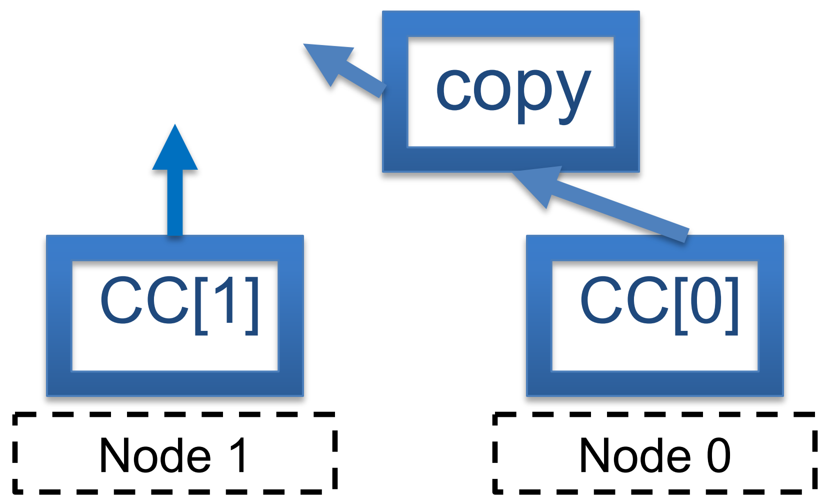


Correctness Independent of Mapping

```
task simulate_circuit(Region[Node] N, Region[Wires] W) :
                ReadWrite(N,W)
{
   …
   calc_currents(piece[0], □ □ □);
   calc_currents(piece[1], □ □ □);
   distribute_charge(piece[0], □ □ □);
   distribute_charge(piece[1], □ □ □);
   …
}
```

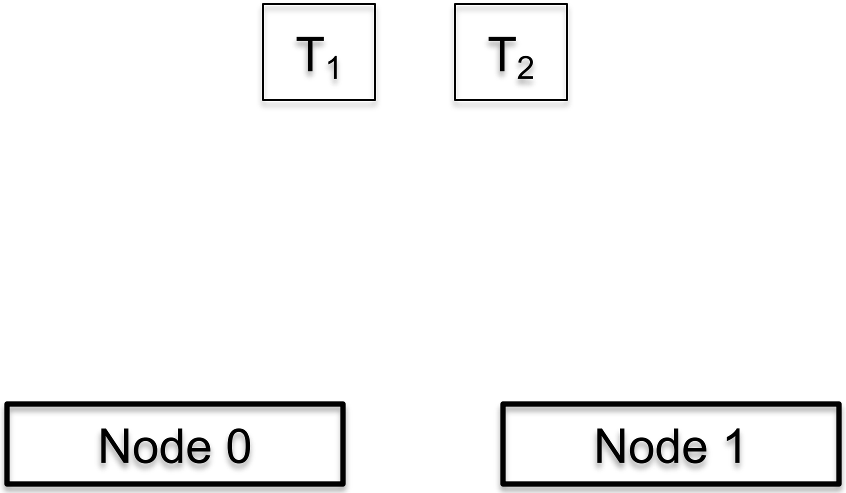


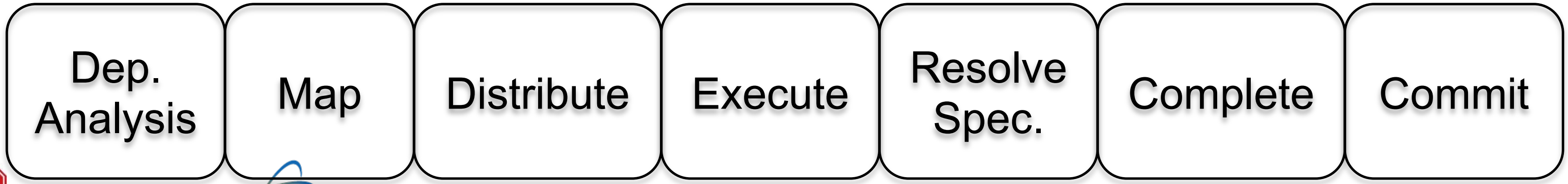

```
task calc_currents(Piece p) :
ReadWrite(p.wires),  Read(p.private, p.shared, p.ghost)

task distribute_charge(Piece p) :
ReadOnly(p.wires), Reduce(p.private, p.shared, p.ghost)
```

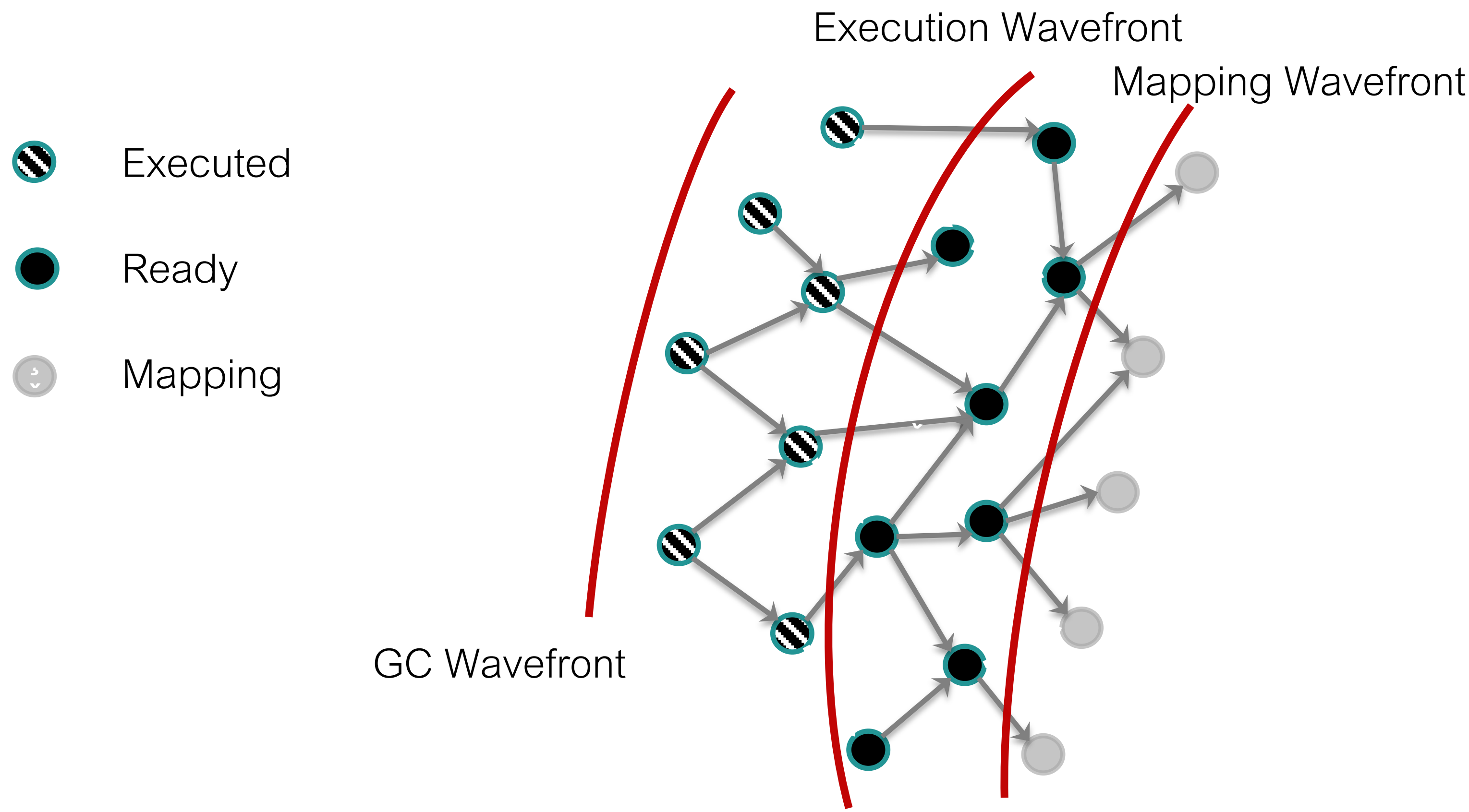


Distribution

- **After tasks are mapped they are distributed to their target nodes**







Dep. Analysis | Map | Distribute | Execute | Resolve Spec. | Complete | Commit

Runtime Summary

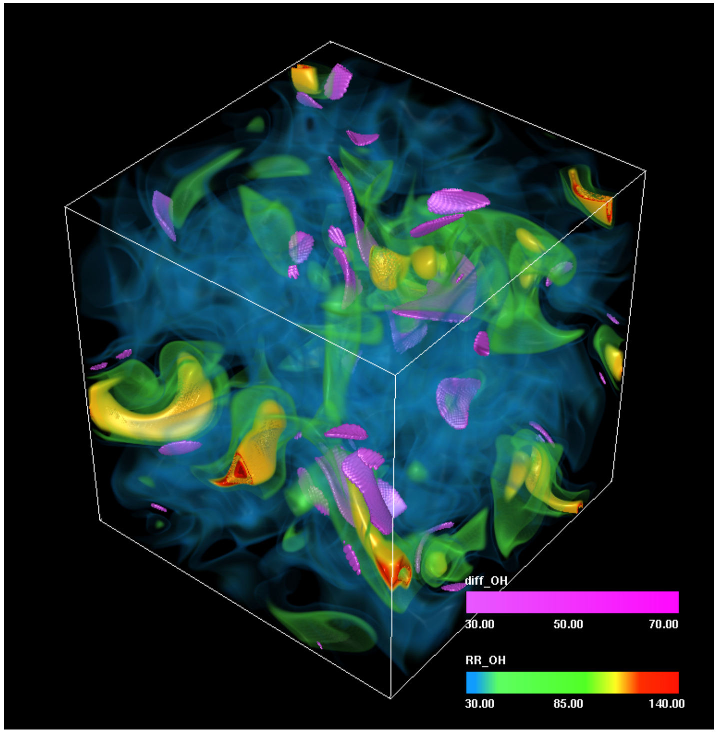
- A distributed hierarchical out-of-order task processor
  - Analogous to hardware processors
- Can exploit parallelism implicitly:
  - Task-, data-, and nested-parallelism
- Runtime builds task graph ahead of execution to hide latency and costs of dynamic analysis
- Decouples mapping decisions from correctness
  - Enables efficient porting and (auto) tuning

Results

S3D: Combustion Simulation

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

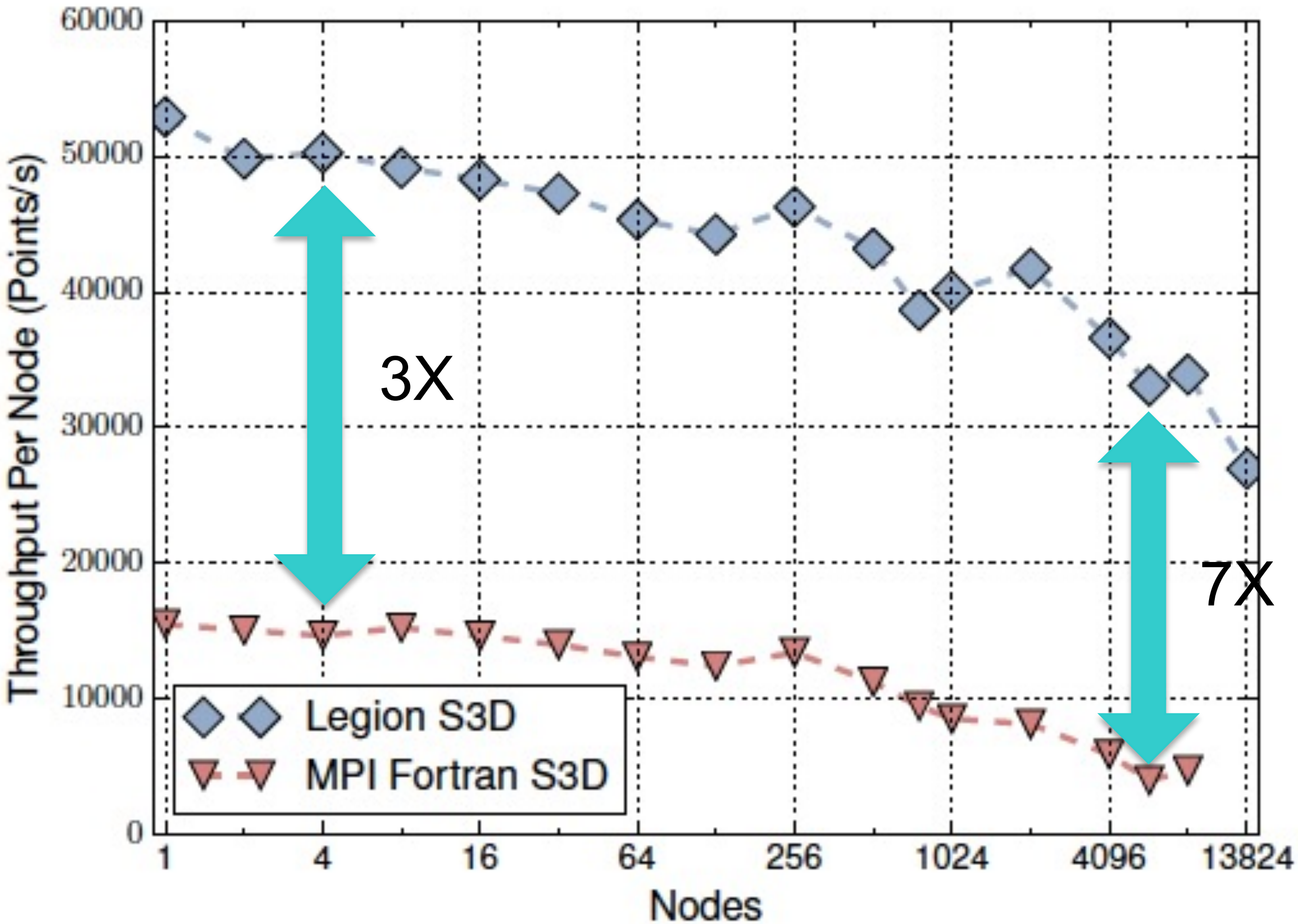
- Two parts
  - Physics
    - Nearest neighbor communication
    - Data parallel
  - Chemistry
    - Local
    - Complex task parallelism
  - Large working sets/task



Recent 3D DNS of auto-ignition with 30-species DME chemistry (Bansal *et al*. 2011)

NVIDIA Los Alamos NATIONAL LABORATORY EST. 1943 SLAC

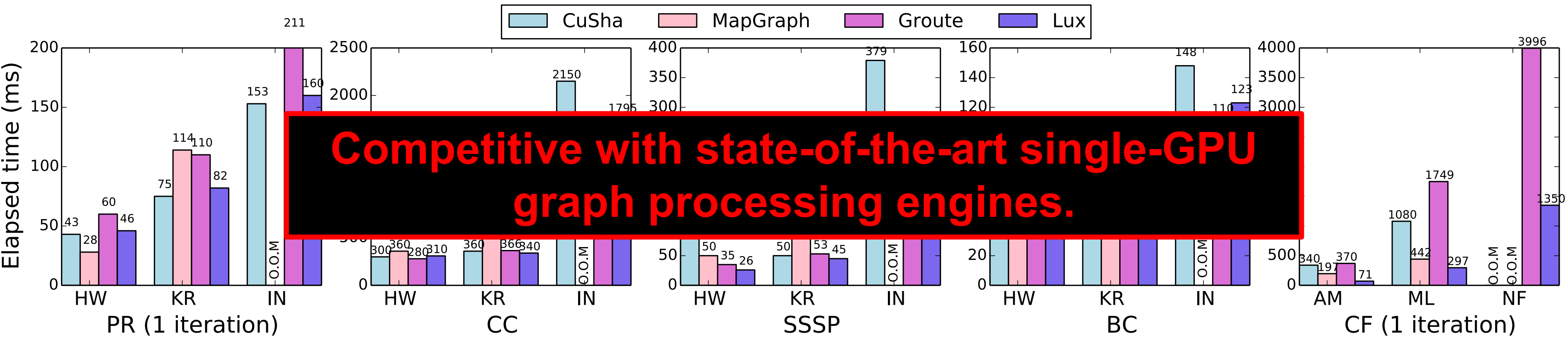
Weak Scaling: PRF on Titan [PLDI14]



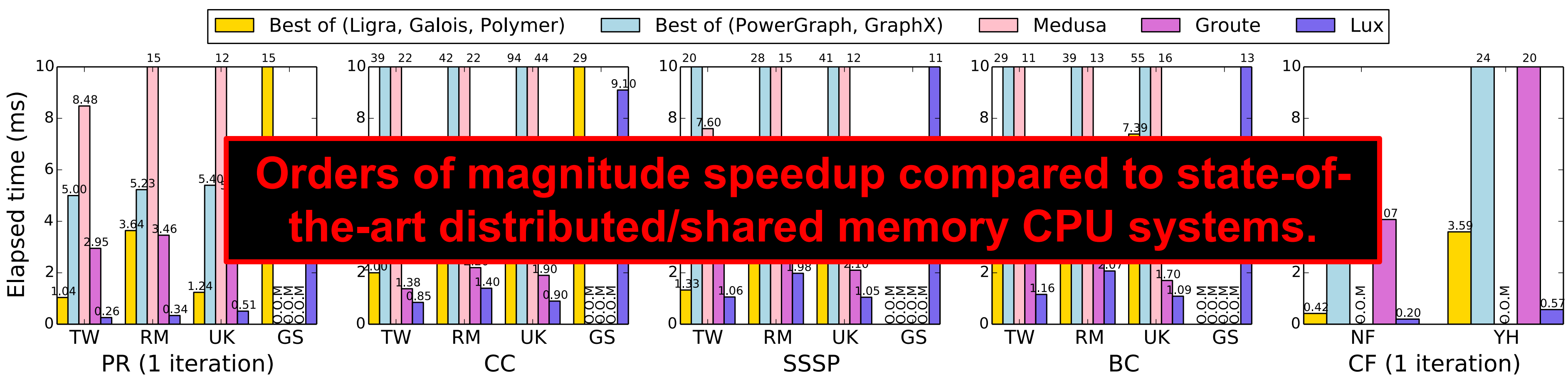
Fast Graph Analytics. [VLDB17]

- **Conventional wisdom:**
  - **Graph processing has trouble taking advantage of distributed memory**
- **High performance graph processing systems are dominated by shared-memory CPU-based systems**
- **Observation: Current GPUs provide much higher memory bandwidth than current CPUs.**

Fast Graph Processing



Competitive with state-of-the-art single-GPU graph processing engines.



Orders of magnitude speedup compared to state-of-the-art distributed/shared memory CPU systems.

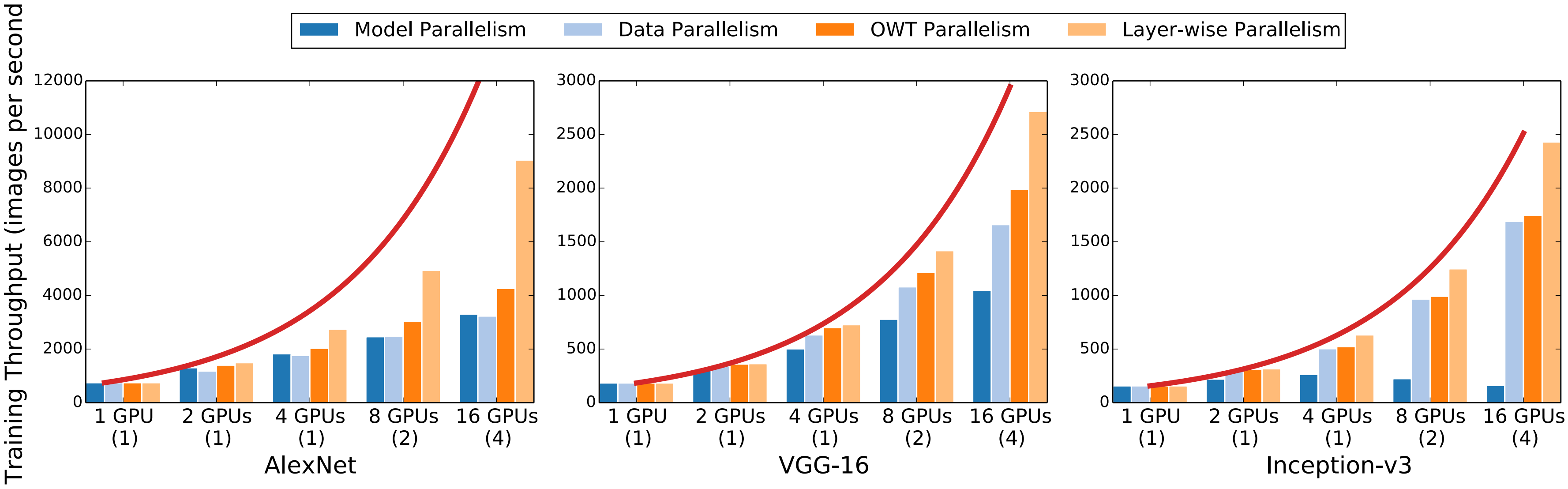
Deep Neural Networks [ICML18]

- In CNNs, data is commonly organized as 4D tensors.
  - tensor = [image, height, width, channel]

- Existing tools parallelize the *image* dimension.

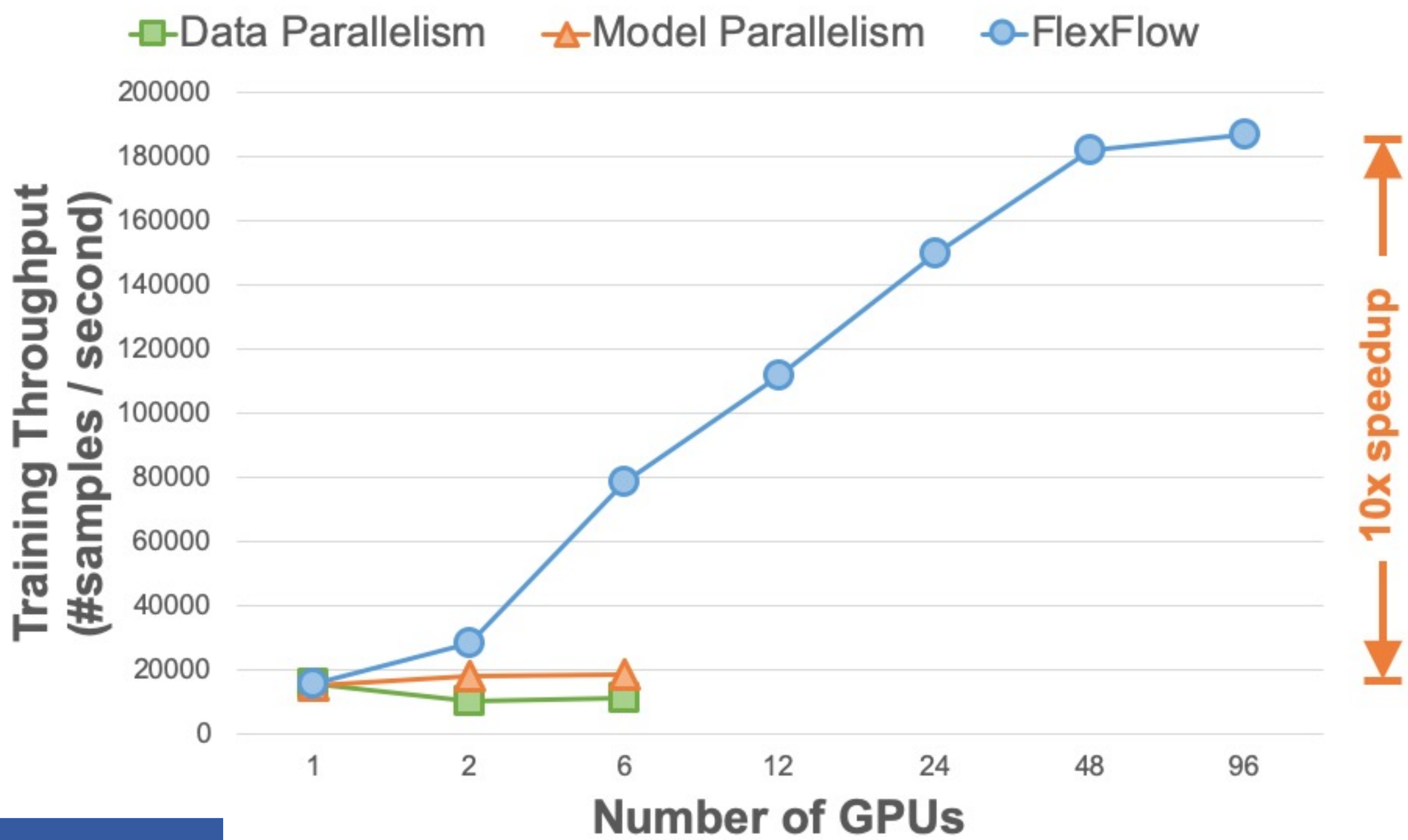
- Idea
  - Explore other parallelizable dimensions
  - Allow each layer to be parallelized differently
  - Automate the search over possible parallelizations

Results



Model Parallelism
Data Parallelism
OWT Parallelism
Layer-wise Parallelism
Training Throughput (images per second
1 GPU (1)
2 GPUs (1)
4 GPUs (1)
8 GPUs (2)
16 GPUs (4)
AlexNet
VGG-16
Inception-v3

DLRM Training Performance



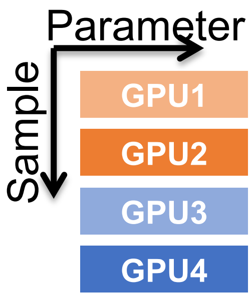
facebook

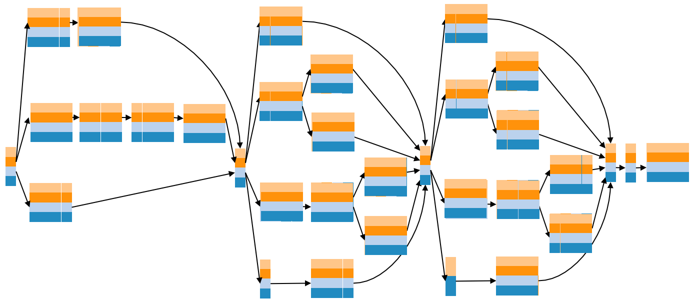
NVIDIA

Los Alamos NATIONAL LABORATORY EST. 1943

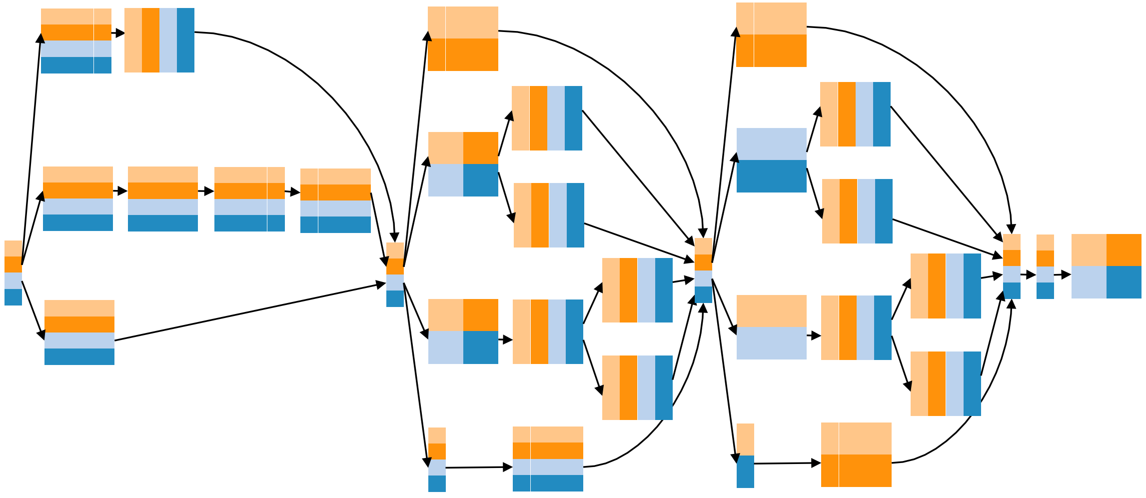
SLAC

Exploiting Multiple Partitions





Data parallelism

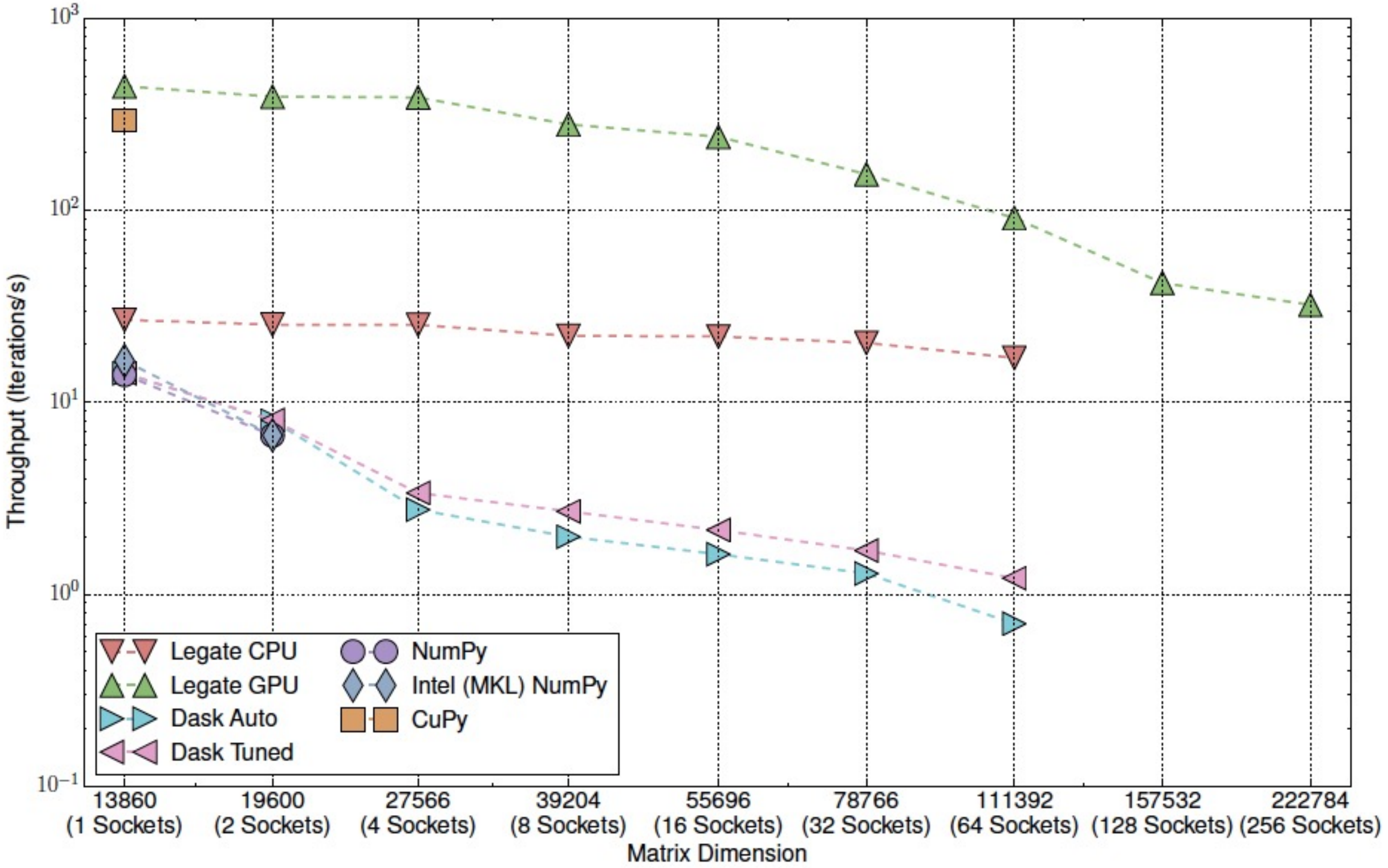


A faster strategy using multiple partitions

Legate NumPy [SC19]

- Legate NumPy is a drop-in replacement for NumPy
  - Implemented on top of Legion
- One line change to use Legate
  - “import legate” instead of “import numpy”
- Now cuNumeric …

Legate Results



Perspectives

Separating Concerns

- Current practice entangles functionality, scheduling, and mapping
  - Heuristics hidden in the runtime system
  - Or exposed ala MPI + OpenMP + CUDA

- Alternative
  - Specify functionality and dependencies first
  - Then focus on mapping and scheduling for a machine

Programmer Productivity

- In the end, it's all about productivity
- How much work is needed to achieve a desired level of performance?
- Legion philosophy
  - Expressive data model, compositionality
  - Requires more initial work from the programmer
  - But makes later stages easier & more flexible
    - E.g., allows easy exploration of alternative mappings

Legion

Legion website: http://legion.stanford.edu