Legion: Programming Heterogeneous, Distributed Parallel Machines

Alex Aiken
Stanford

Joint work involving LANL, NVIDIA & Stanford

Modern Supercomputers

- **Heterogeneity**
  - **Processor kinds**
  - **Relative performance**
- **Distributed Memory**
  - **Non-uniform**
  - **Distinct from processors**
- **Growing disparities**
  - **FLOPS >> bandwidth**
  - **bandwidth >> latency**

Programming System Goals

High Performance

We must be fast

Performance Portability

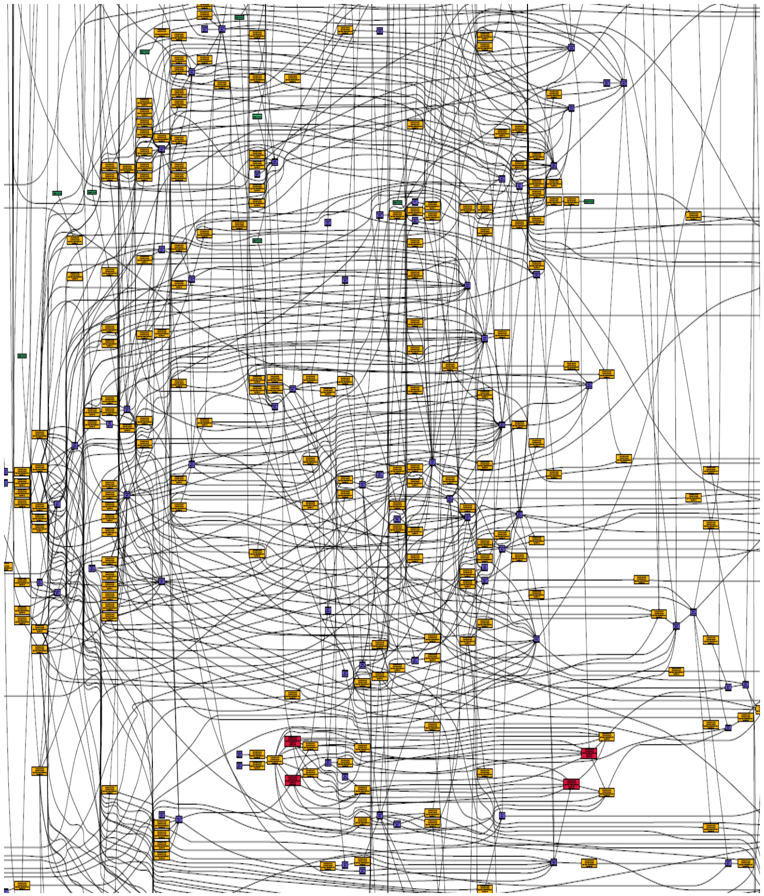
Across many kinds of machines and over many generations

Programmability

Sequential semantics, parallel execution

Can We Fulfill These Goals Today?

Yes … at great cost:



Task graph for one time step on one node…

… of a mini-app

Do you want to schedule that graph?
(High Performance)

Do you want to re-schedule that graph for every new machine?
(Performance Portability)

Do you want to be responsible for generating that graph?
(Programmability)

Today: programmer's responsibility

Tomorrow: programming system's responsibility

The Crux

- **How do we describe the data?**
  - **Most programming systems focus on control**
  - **Minimal facilities for organization/structure of data**

- **Why?**

- **Answer: Solve the *aliasing problem***
  - **Can two references refer to the same data?**

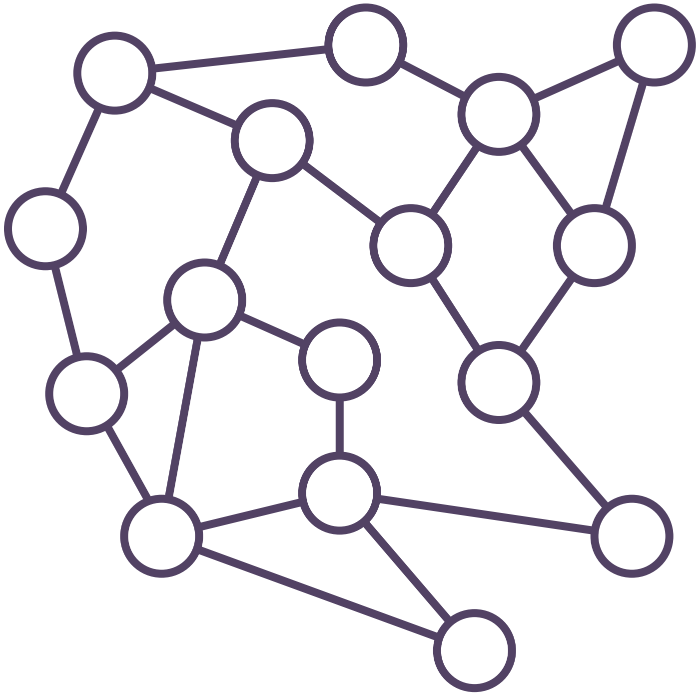
- **Answer: Decouple naming data from layout/location**

Legion Approach

- Capture the structure of program data
- Decouple specification from *mapping*
- Asynchronous tasking
- Automate
  - data movement
  - parallelism discovery
  - synchronization
  - hiding long latency

Legion Programming Model

Example: Circuit Simulation



Example: Circuit Simulation

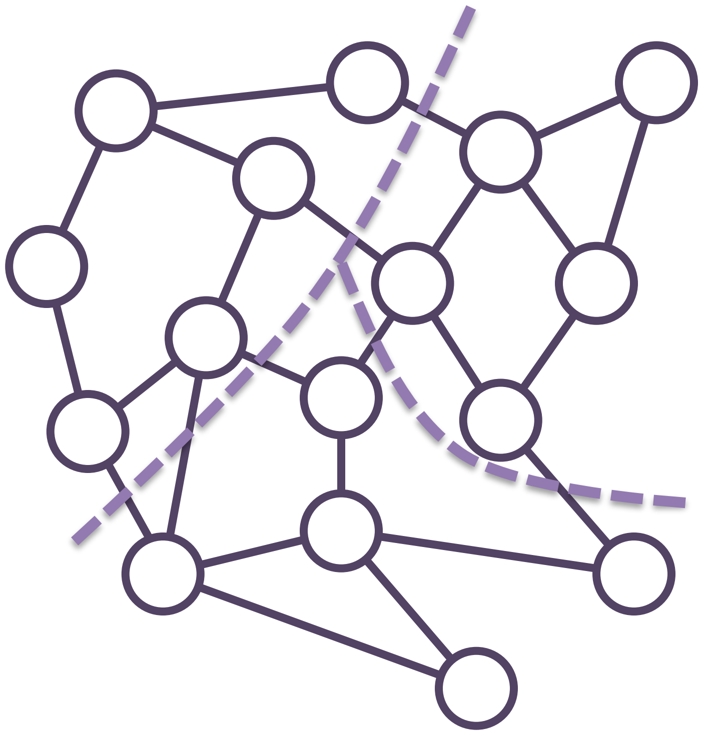
```
task simulate_circuit(Region[Node] N, Region[Wires] W)
{

}
```

Tasks are the unit of parallel execution.

Logical regions are (typed) collections

Logical:
- no implied layout
- no implied location

Partitioning

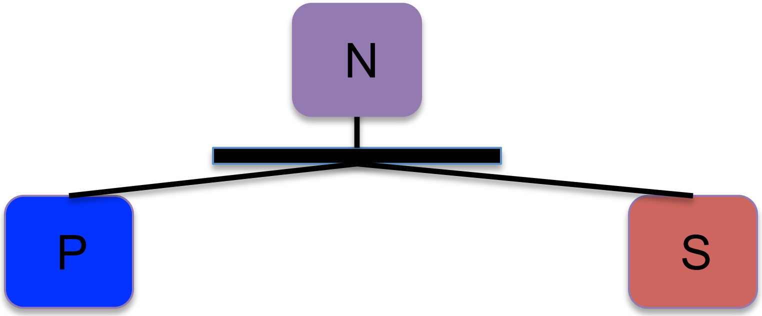


Partitioning

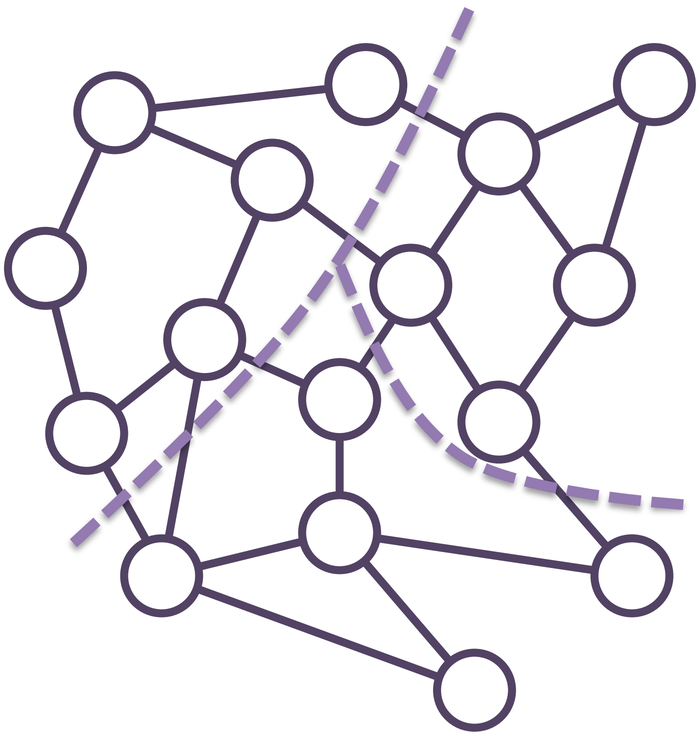
task simulate_circuit(Region[Node] N, Region[Wires] W)

{

[P, S] = partition(ps_map, N)

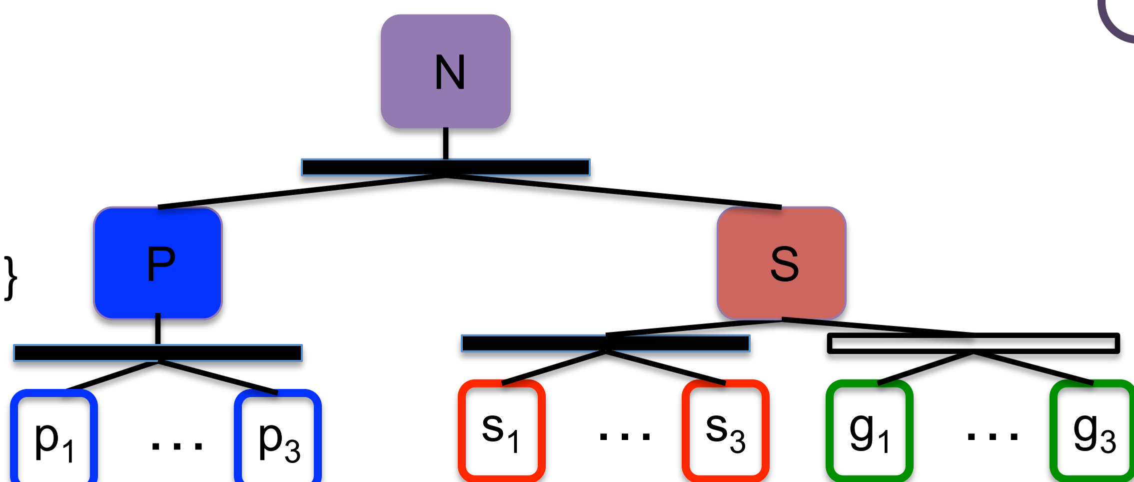


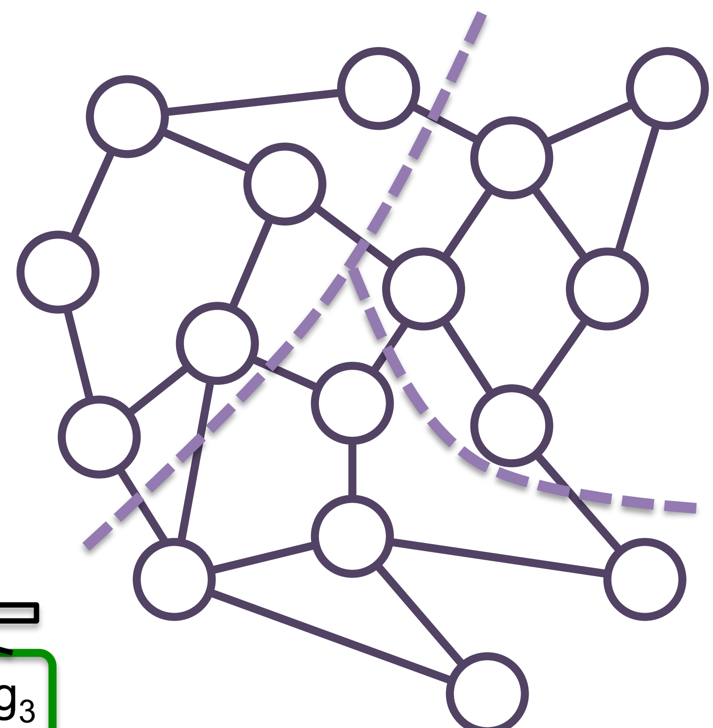
}



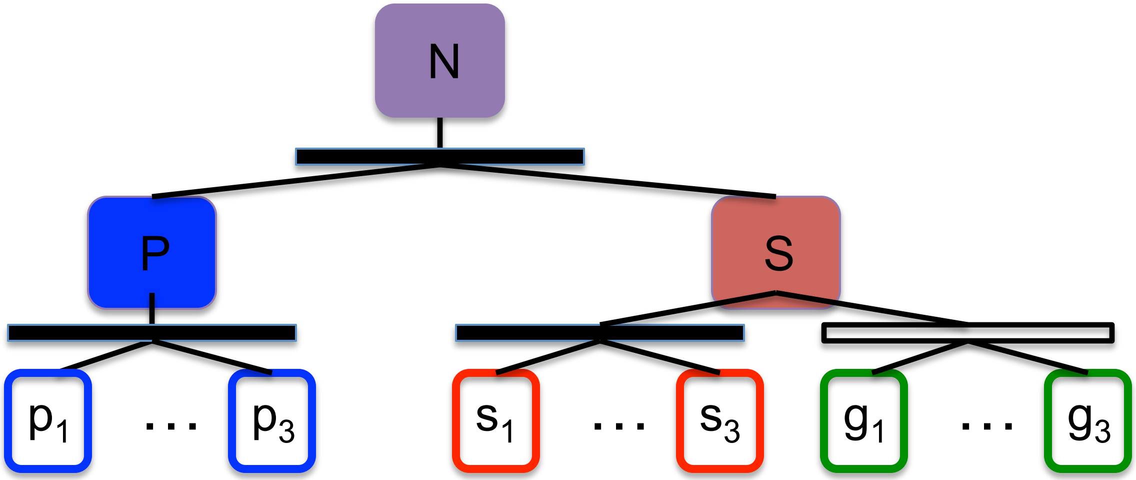
Partitioning

```
task simulate_circuit(Region[Node] N, Region[Wires] W)

{
    Array[Region[Node]] private, shared, ghost;
    …
    [ P, S ] = partition(ps_map, N)
    private = partition(private_map, P)
    shared = partition(shared_map, S)
    ghost = partition(ghost_map, S)
}
```

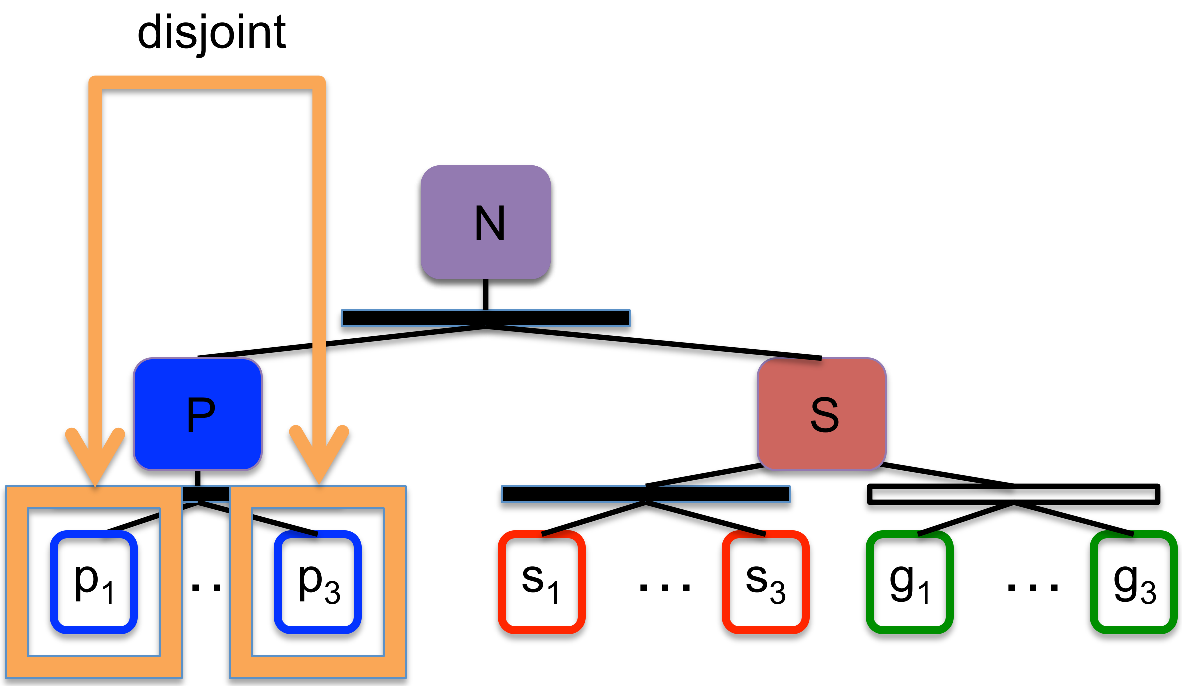




Region Trees

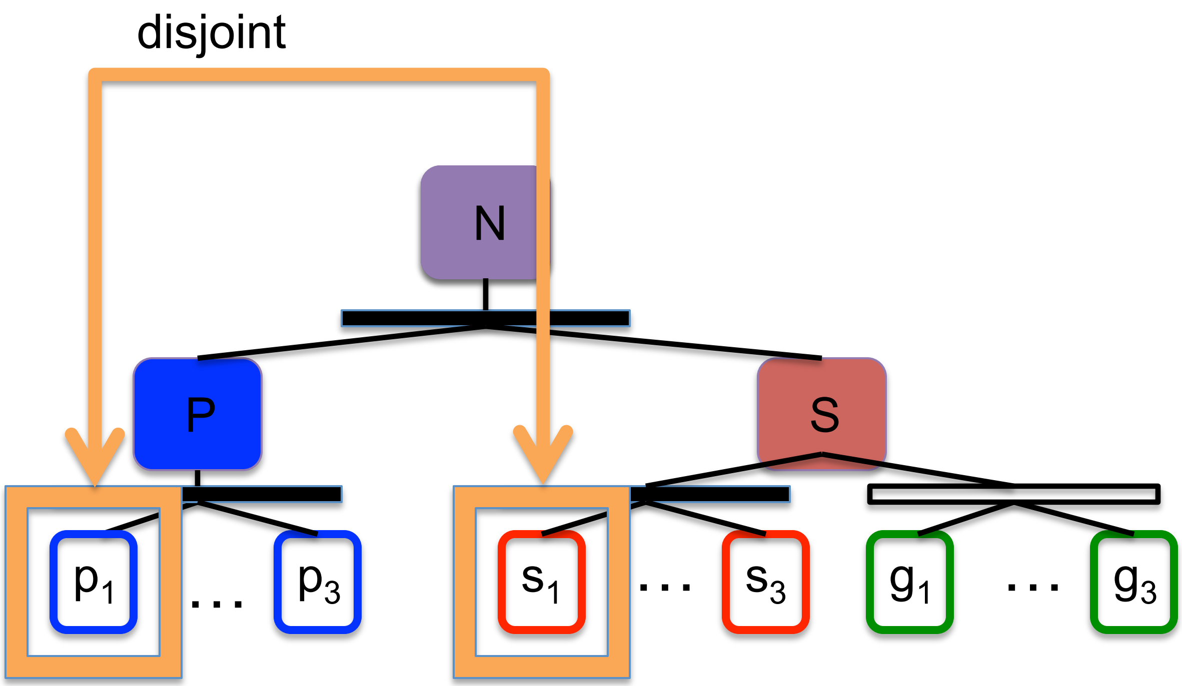


Region Trees



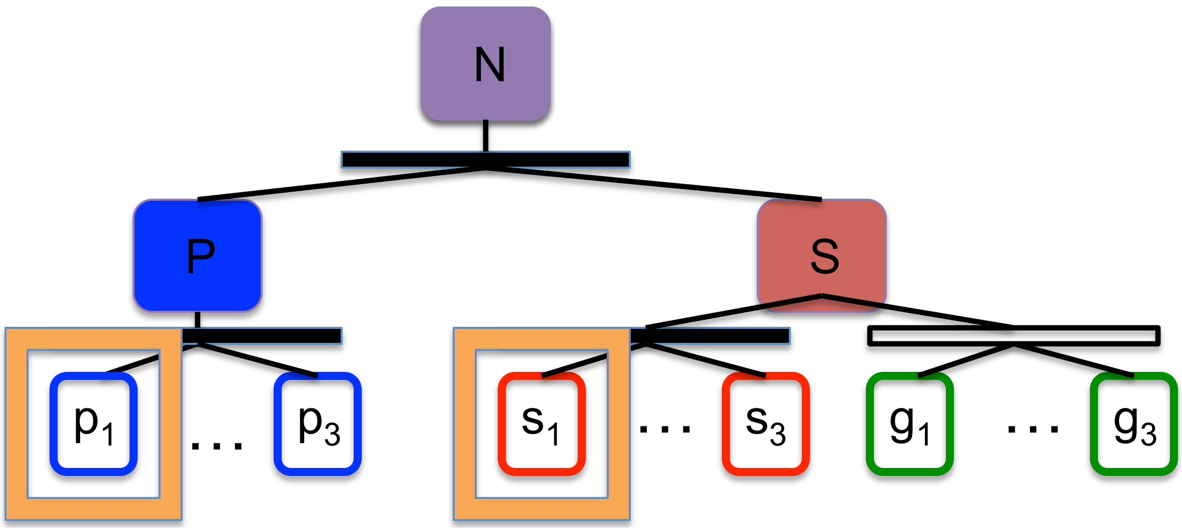
Locality
Independence

Region Trees



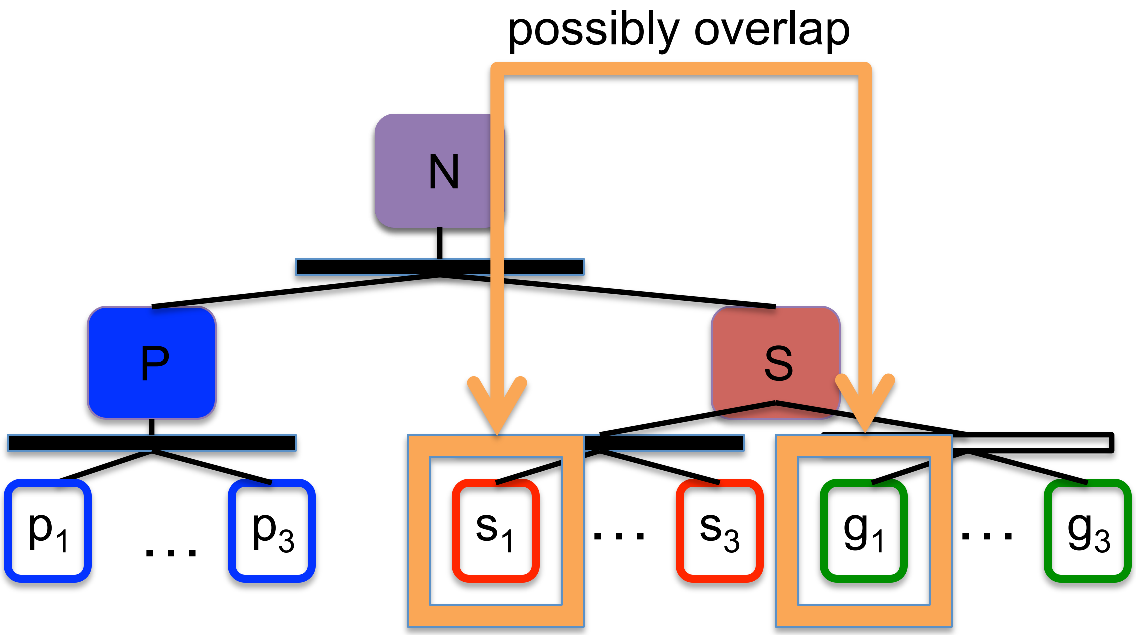
Locality
Independence

Region Trees



Locality
Independence
Aliasing

Region Trees



Locality

Independence

Aliasing

Legion Tasks

```
task simulate_circuit(Region[Node] N, Region[Wires] W) :

{
    …
    calc_currents(piece[0]);
    calc_currents(piece[1]);
    distribute_charge(piece[0]);
    distribute_charge(piece[1]);
    …
}

task calc_currents(Piece p) :

task distribute_charge(Piece p) :
```

subtasks

Legion Tasks

```
task simulate_circuit(Region[Node] N, Region[Wires] W) :
            N, W
{
    …
    calc_currents(piece[0], p0, s0, g0);
    calc_currents(piece[1], p1, s1, g1);
    distribute_charge(piece[0], p0, s0, g0);
    distribute_charge(piece[1], p1, s1, g1);
    …
}

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

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

A task must declare the regions it will use.

Subtask containment: A subtask can only use (sub)regions accessible to its parent task.

Tasks appear to execute in program order.

Interference

Informal definition: Two tasks T_1 and T_2 are non-interfering $T_1 \# T_2$ if there is no dependence between them on their region arguments.

If $T_1 \# T_2$ then T_1 and T_2 can execute in parallel.

Execution Model

```
task simulate_circuit(Region[Node] N, Region[Wires] W) :
{
    ...
    calc_currents(piece[0], p0, s0, g0);
    calc_currents(piece[1], p1, s1, g1);
    distribute_charge(piece[0], p0, s0, g0);
    distribute_charge(piece[1], p1, s1, g1);
    ...
}
```

Interferes?

Interferes?

Tasks are issued in program order.

Privileges

task simulate_circuit(Region[Node] N, Region[Wires] W) :

N, W

ReadWrite(N,W)

{

...

calc_currents(piece[0], p_0, s_0, g_0);

calc_currents(piece[1], p_1, s_1, g_1);

distribute_charge(piece[0], p_0, s_0, g_0);

distribute_charge(piece[1], p_1, s_1, g_1);

...

}

task calc_currents(Piece p) :

p.private, p.shared, p.ghost, p.wires

ReadWrite(p.wires), Read(p.private, p.shared, p.ghost)

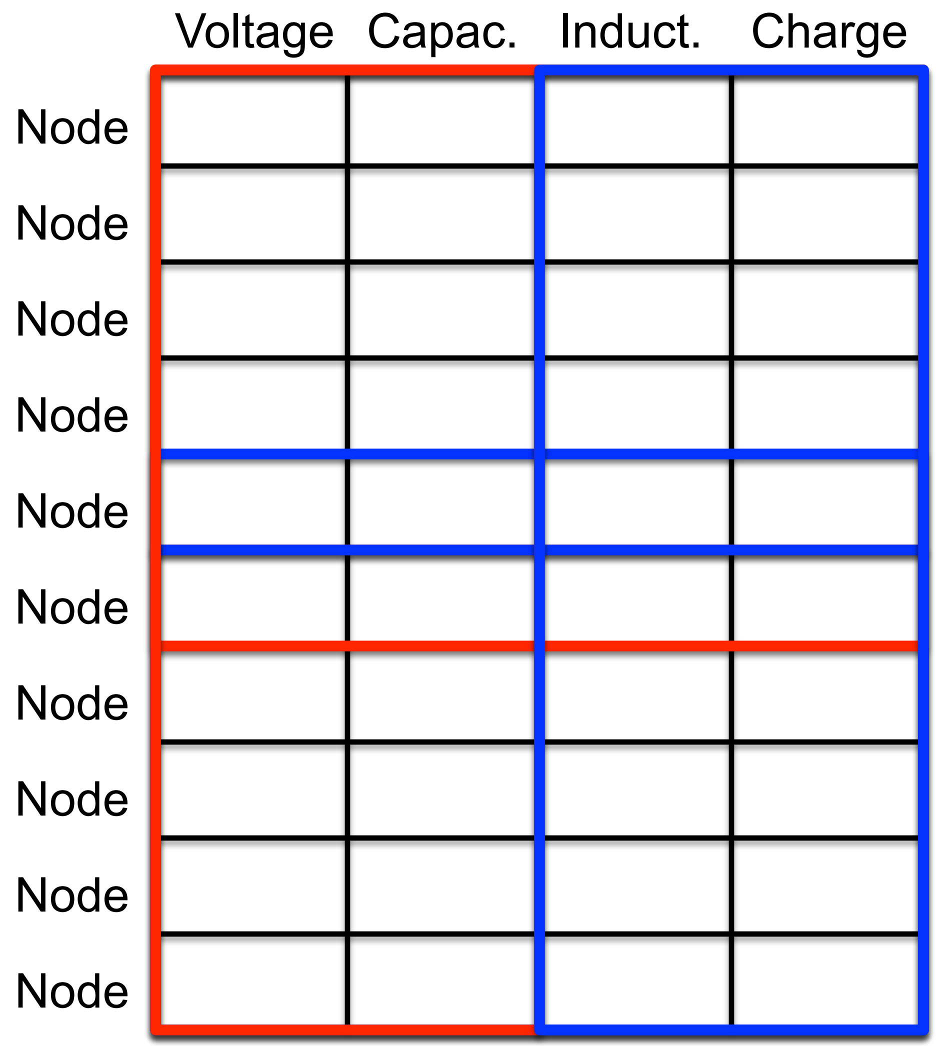
task distribute_charge(Piece p) :

p.private, p.shared, p.ghost, p.wires

ReadOnly(p.wires), Reduce(p.private, p.shared, p.ghost)

Non-Interference Dimensions

- Several dimensions of # operator
  - Entries (rows)
  - Privileges
  - Fields (columns)
- Logical regions are a relational data model
  - Partitioning is selection (σ)
  - Field-slicing is projection (π)
  - Don’t support all relational operators

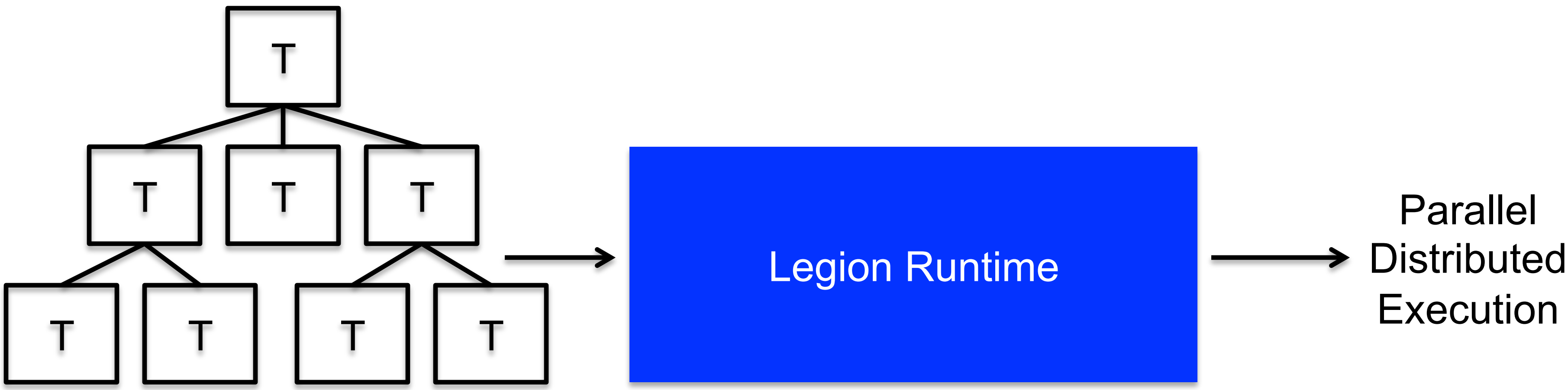


Legion Summary

- Logical regions: a relational data model
  - Support partitioning and slicing
  - Convey locality, independence, aliasing
- Implicit task parallelism
  - Task may have arbitrary sub-tasks
  - Tasks declare region usage including privileges and fields
- Tasks appear to execute in program order
  - Execute in parallel when non-interference established
- Machine independent specification of application

Legion Runtime System

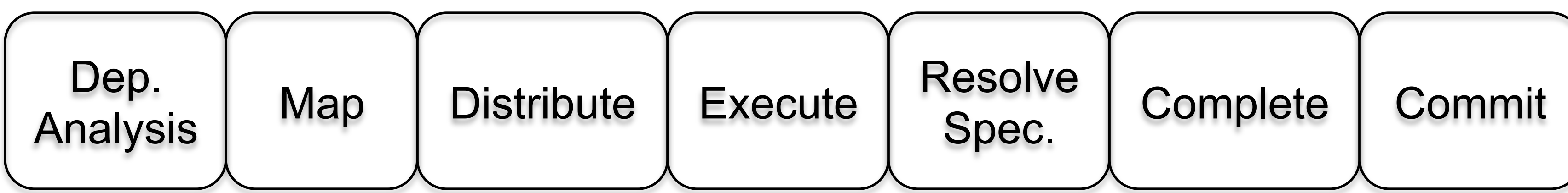
Legion Runtime System



Parent Task

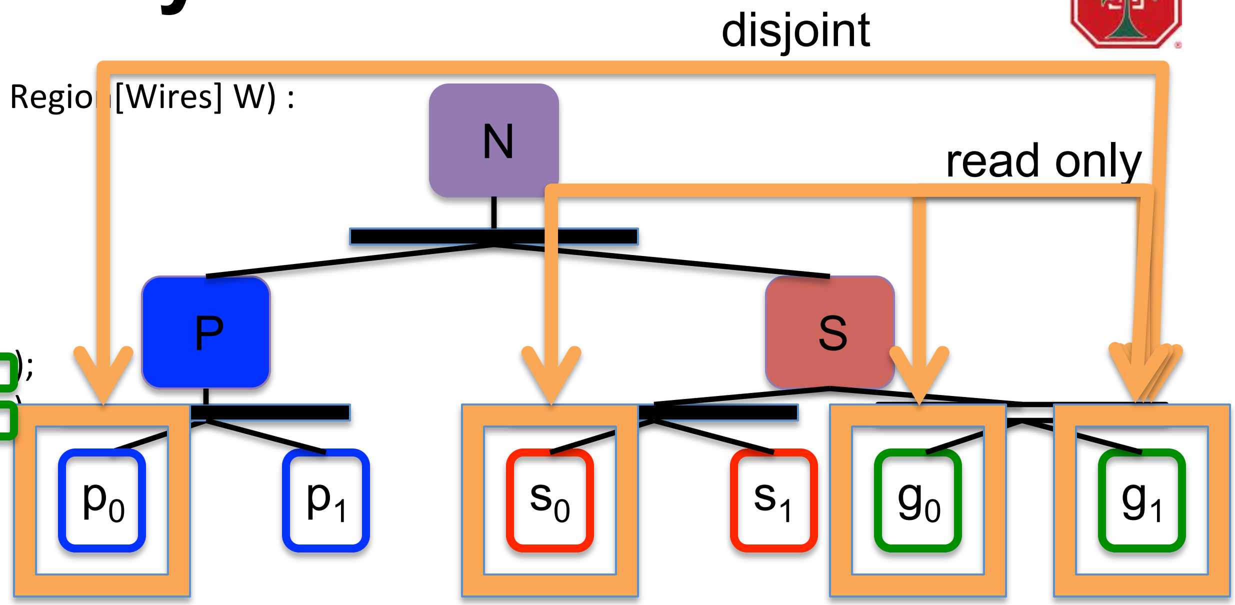
t_0: r_5, r_7
t_1: r_0, r_2
t_2: r_1, r_2
t_3: r_4, r_6

Tasks : Regions :: Instructions : Registers



A Distributed Hierarchical Out-of-Order Task Processor

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] );
   ...
}
```

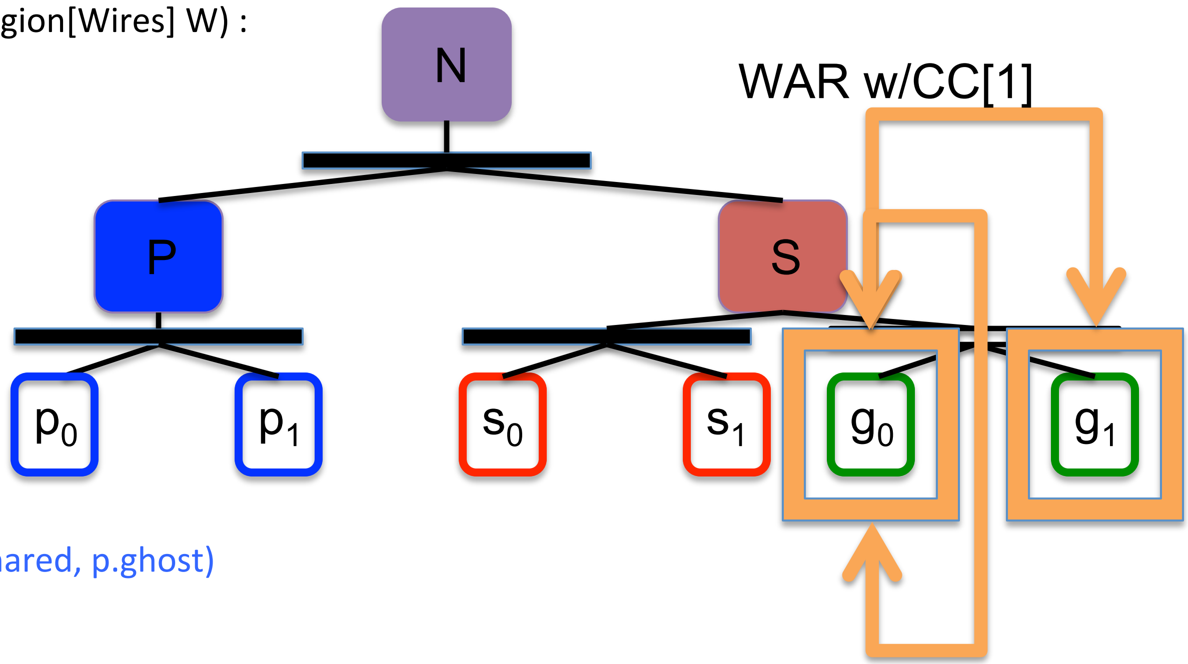
task calc_currents(Piece p) :
ReadWrite(p.wires), Read(p.private, p.shared, p.ghost)

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

CC[0]

Dependence Analysis

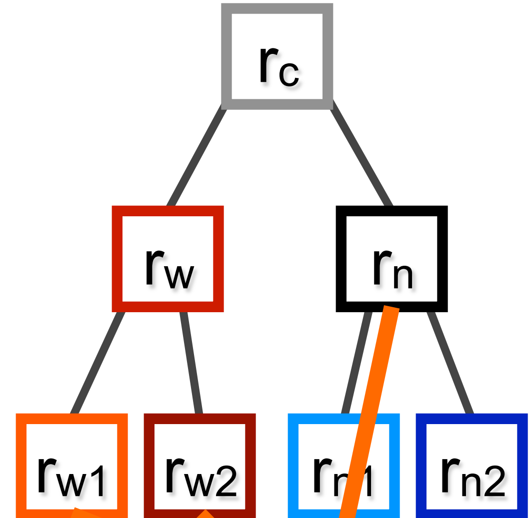
```
task simulate_circuit(Region[Node] N, Region[Wires] W) :
                ReadWrite(N,W)
{
    ...
    calc_currents(piece[0],□□□);
    calc_currents(piece[1],□□□);
→   distribute_charge(piece[0],□□□);
    distribute_charge(piece[1],□□□);
    ...
}

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

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

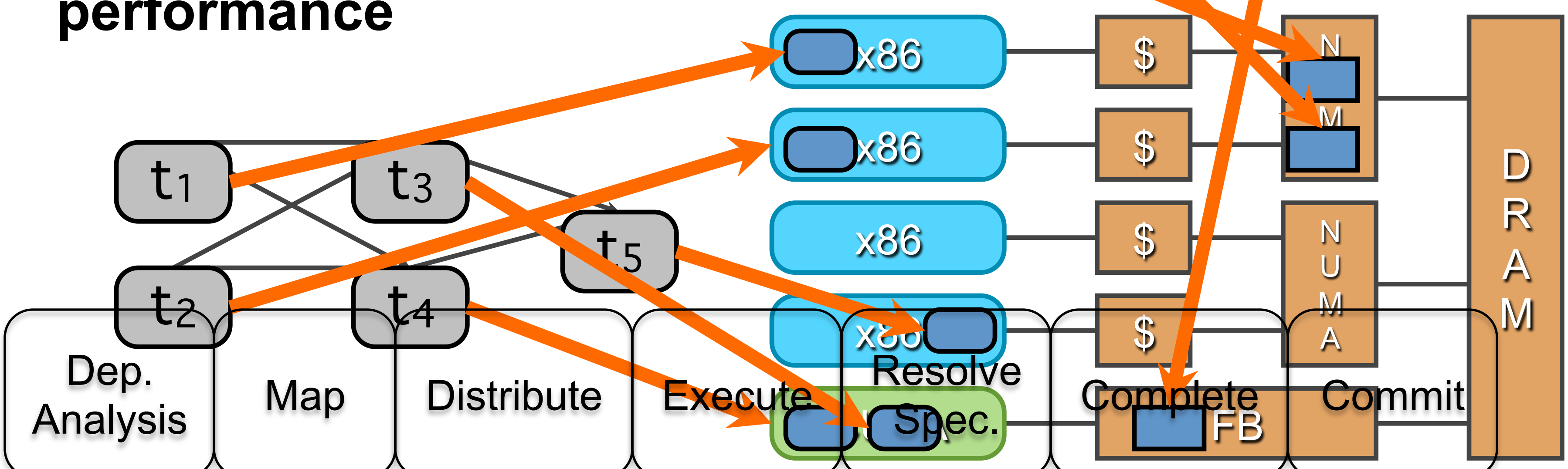


read-after-write of wires w/CC[0]

Mapping Interface

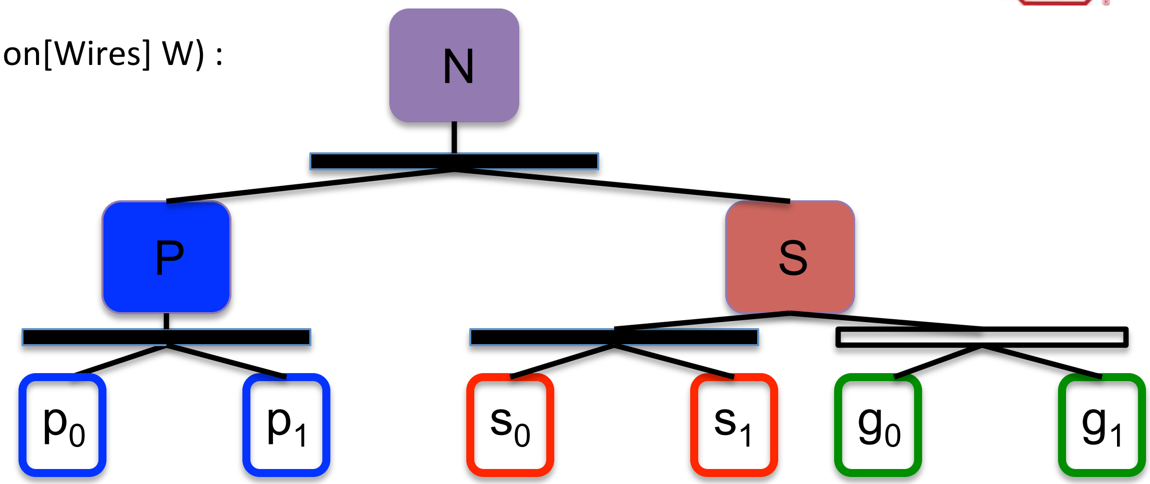
- **Programmer selects:**
  - **Where tasks run**
  - **Where regions are placed**
- **Mapping computed dynamically**
- **Decouple correctness from performance**





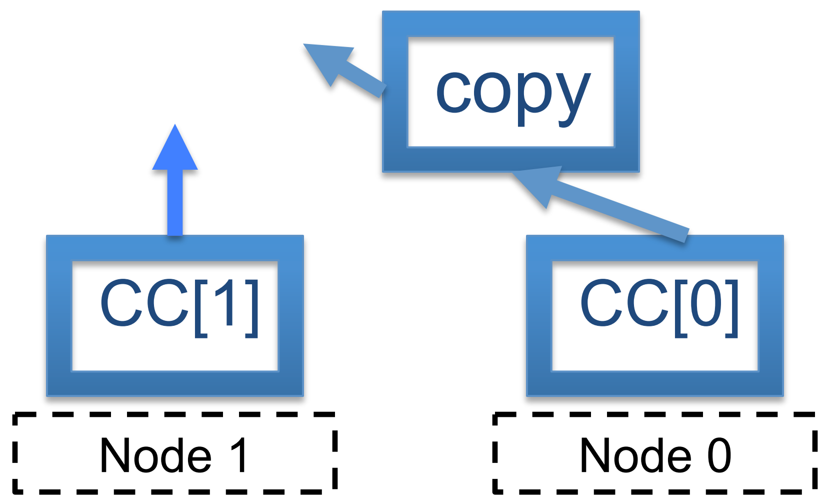
Correctness Independent of Mapping

```
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],   );
   …
}
```



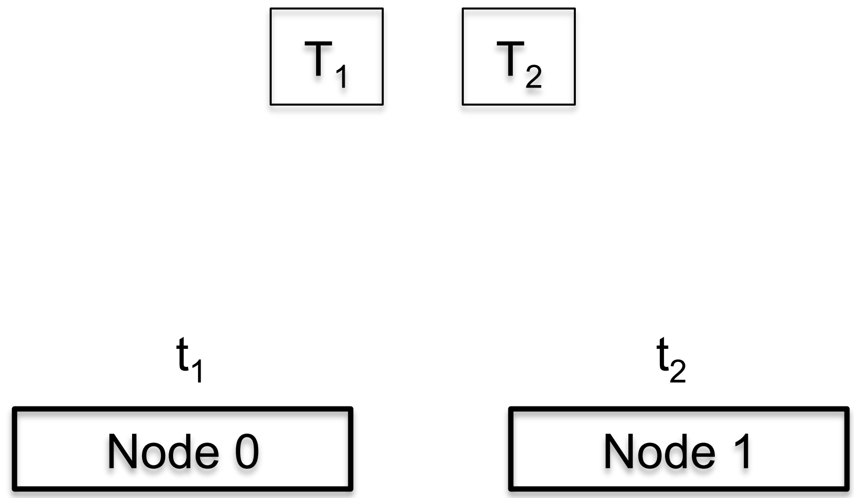
task calc_currents(Piece p) :
ReadWrite(p.wires), Read(p.private, p.shared, p.ghost)

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

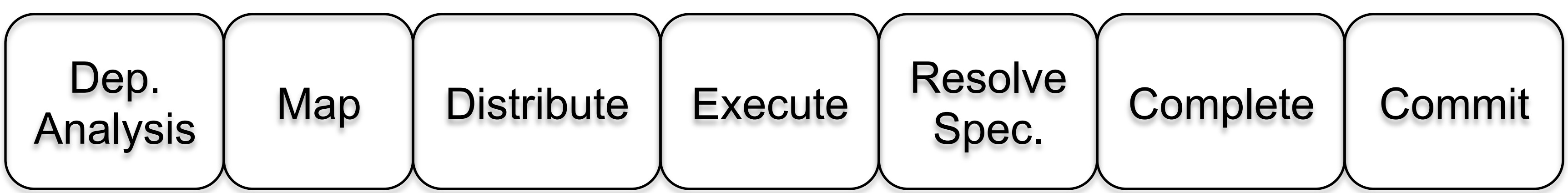


Distribution

- **After tasks are mapped they are distributed to target node**
- **Task execution can generate sub-tasks**
- **Do we need inter-node dependence checks?**



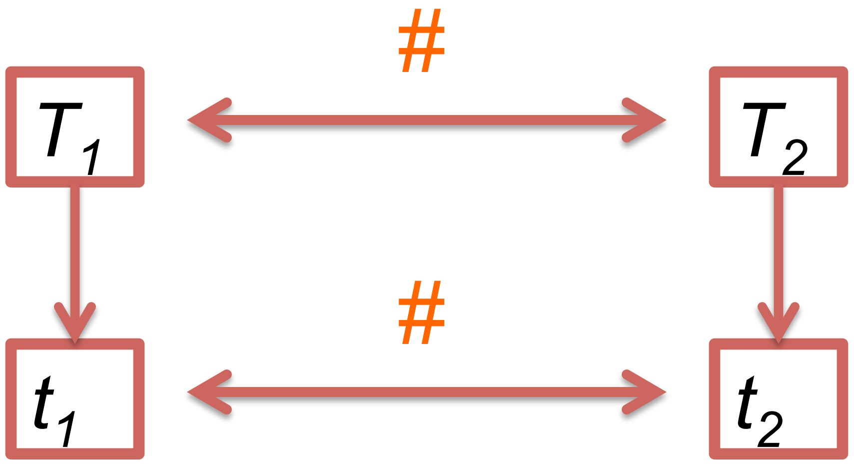
Subtask containment:
A subtask can only use (sub)regions accessible to its parent task.



Independence Theorem

Let t_1 be a subtask of T_1 and t_2 be a subtask of T_2. Then

$$T_1 \# T_2 \Rightarrow t_1 \# t_2$$



Independence Theorem

Let t_1 be a subtask of T_1 and t_2 be a subtask of T_2. Then

$$T_1 \# T_2 \Rightarrow t_1 \# t_2$$

Proof: Use subtask containment.

Observation: It is sufficient to test interference only of sibling tasks.

Note: Similar property holds in functional languages, but it holds in Legion even though we may imperatively mutate regions.

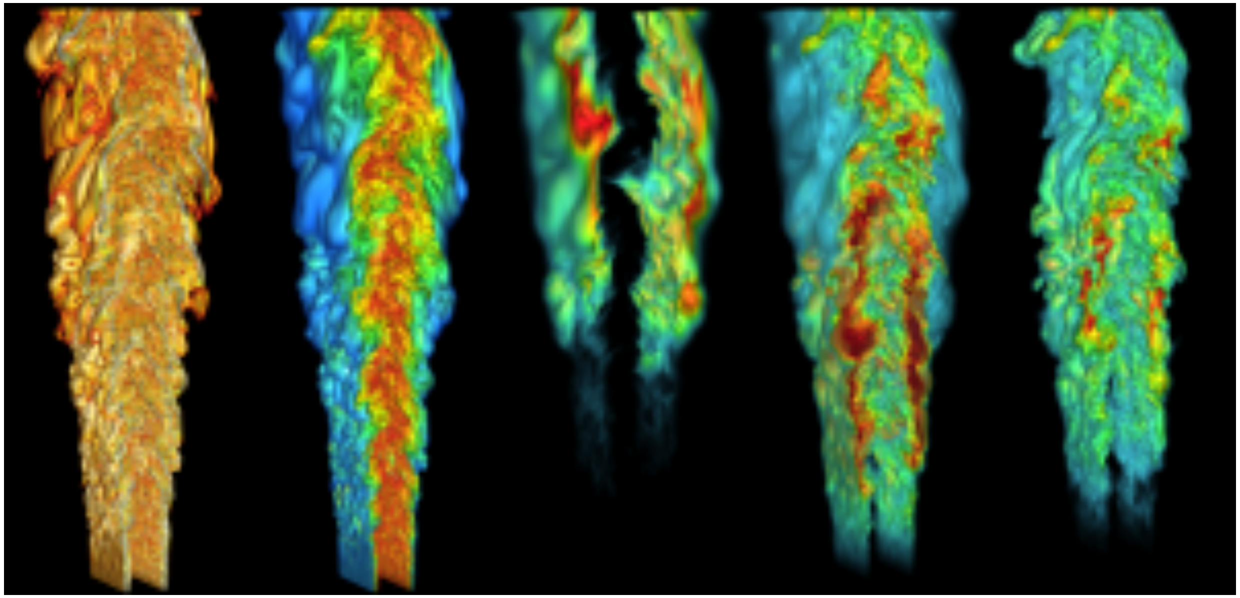
Runtime Summary

- 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

A Real Application: S3D

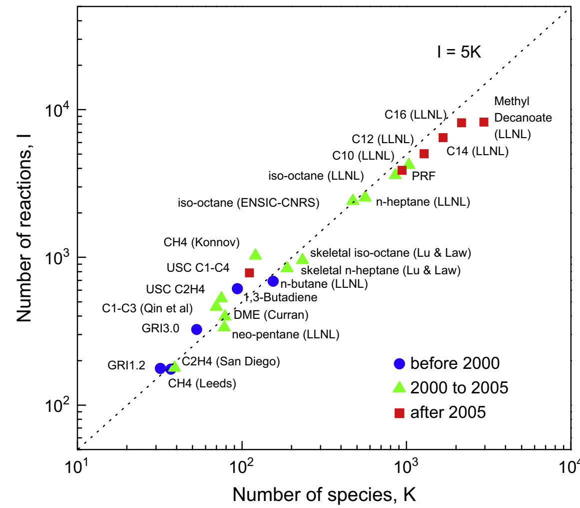
S3D

- Production combustion simulation
- Written in ~200K lines of Fortran
- Direct numerical simulation using explicit methods

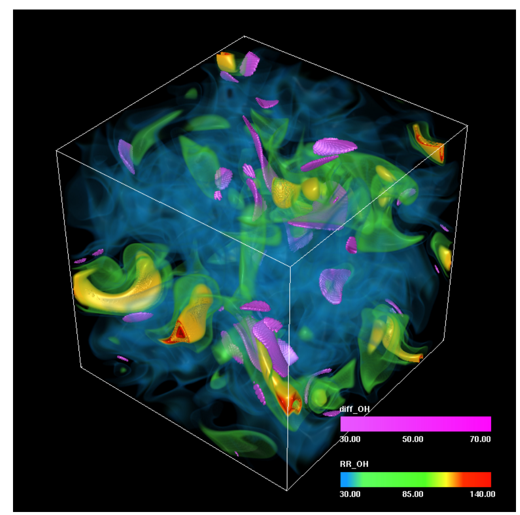


S3D Versions

- Supports many chemical mechanism
  - DME (30 species)
  - Heptane (52 species)
- Fortran + MPI
  - Vectorizes well
  - MPI used for multi-core
- “Hybrid” OpenACC
  - Recent work by Cray/Nvidia/DoE
- Legion interoperates with MPI



From Lu and Law, PECS, 2009



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

Parallelism in S3D

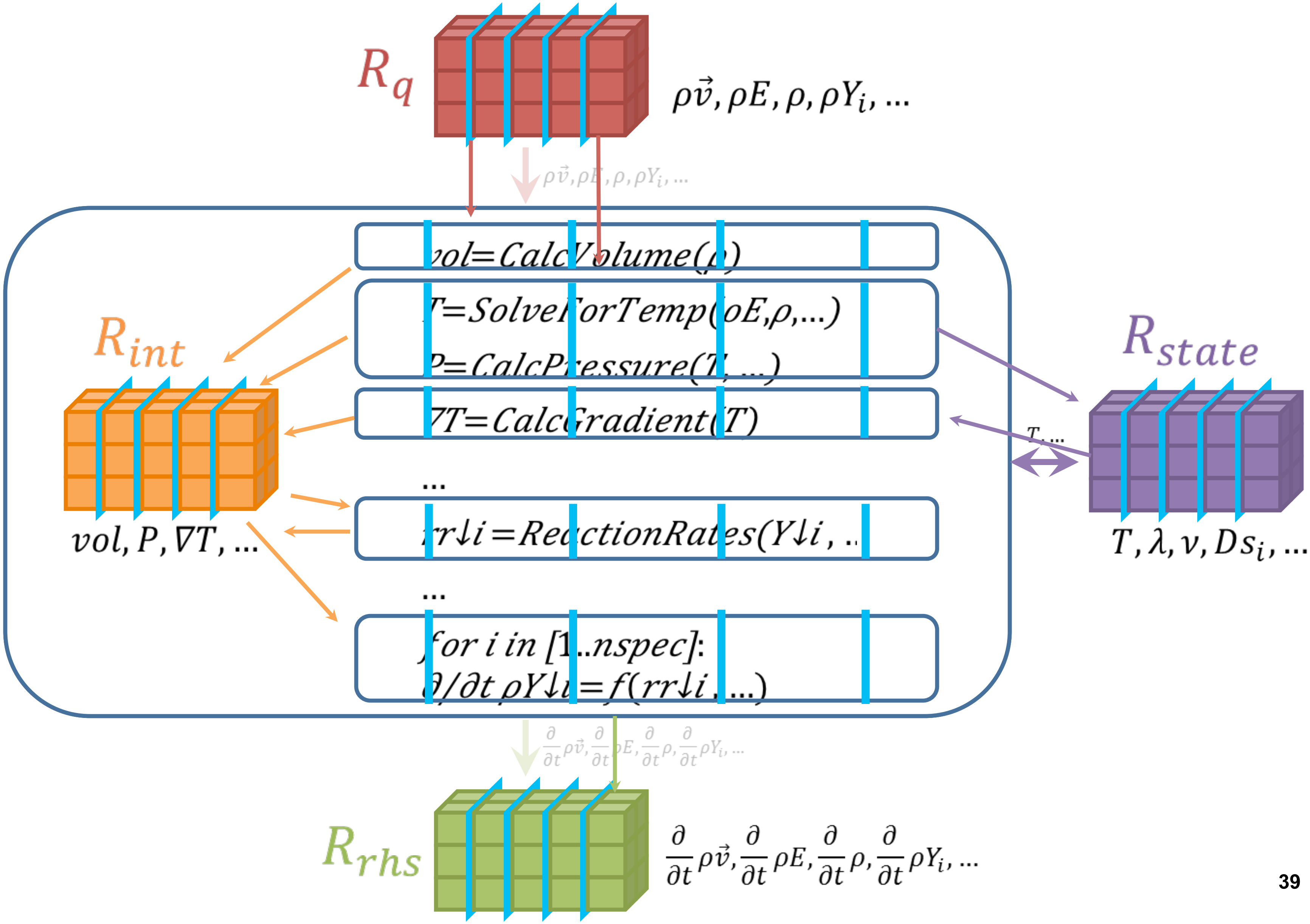
- Data is large 3D cartesian grid of cells

- Typical per-node subgrid is 48^3 or 64^3 cells
  - Nearly all kernels are per-cell
  - Embarrassingly data parallel

- Hundreds of tasks
  - Significant task-level parallelism

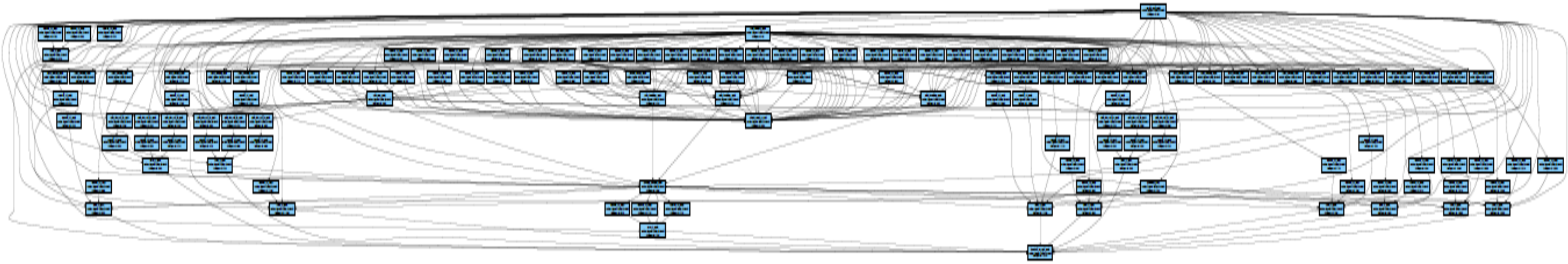
- Except…
  - Computational intensity is low
  - Large working sets per cell (1000s of temporaries)
  - Performance limiter is data, not compute

S3D Tasks in Legion

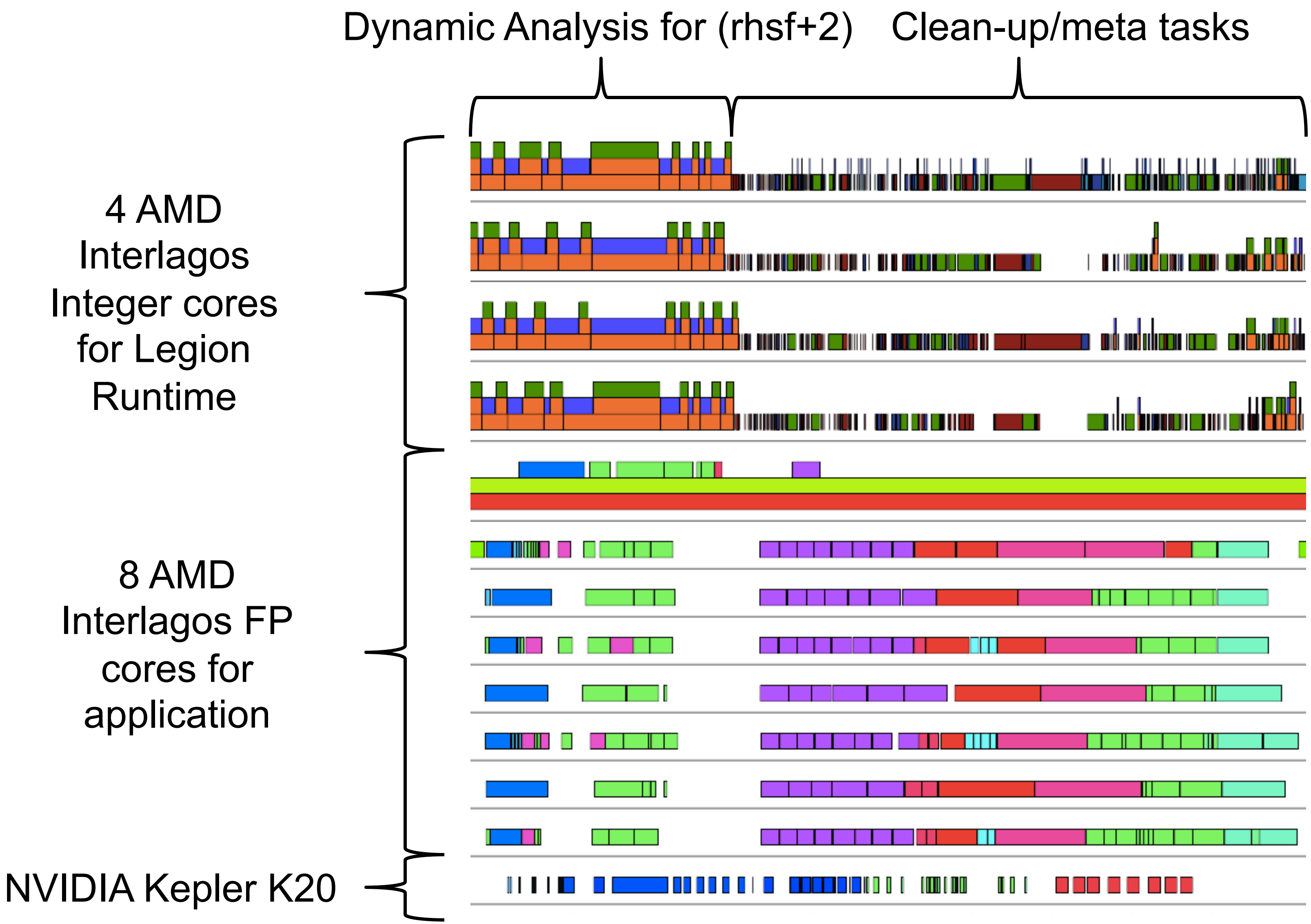


S3D Task Parallelism

- One call to Right-Hand-Side-Function (RHSF) as seen by the Legion runtime
  - Called 6 times per time step by Runge-Kutta solver
  - Width == task parallelism
  - H2 mechanism (only 9 species)
  - Heptane (52 species) is significantly wider
- Manual task scheduling would be difficult!

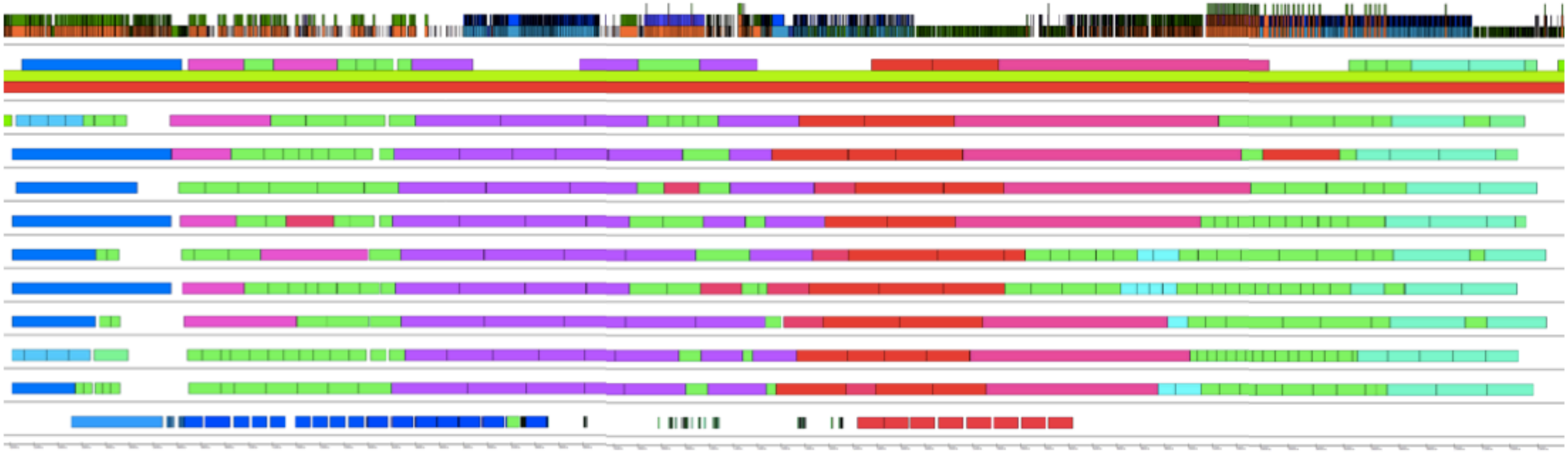


Mapping for Heptane 48^3



Heptane Mapping for 96^3

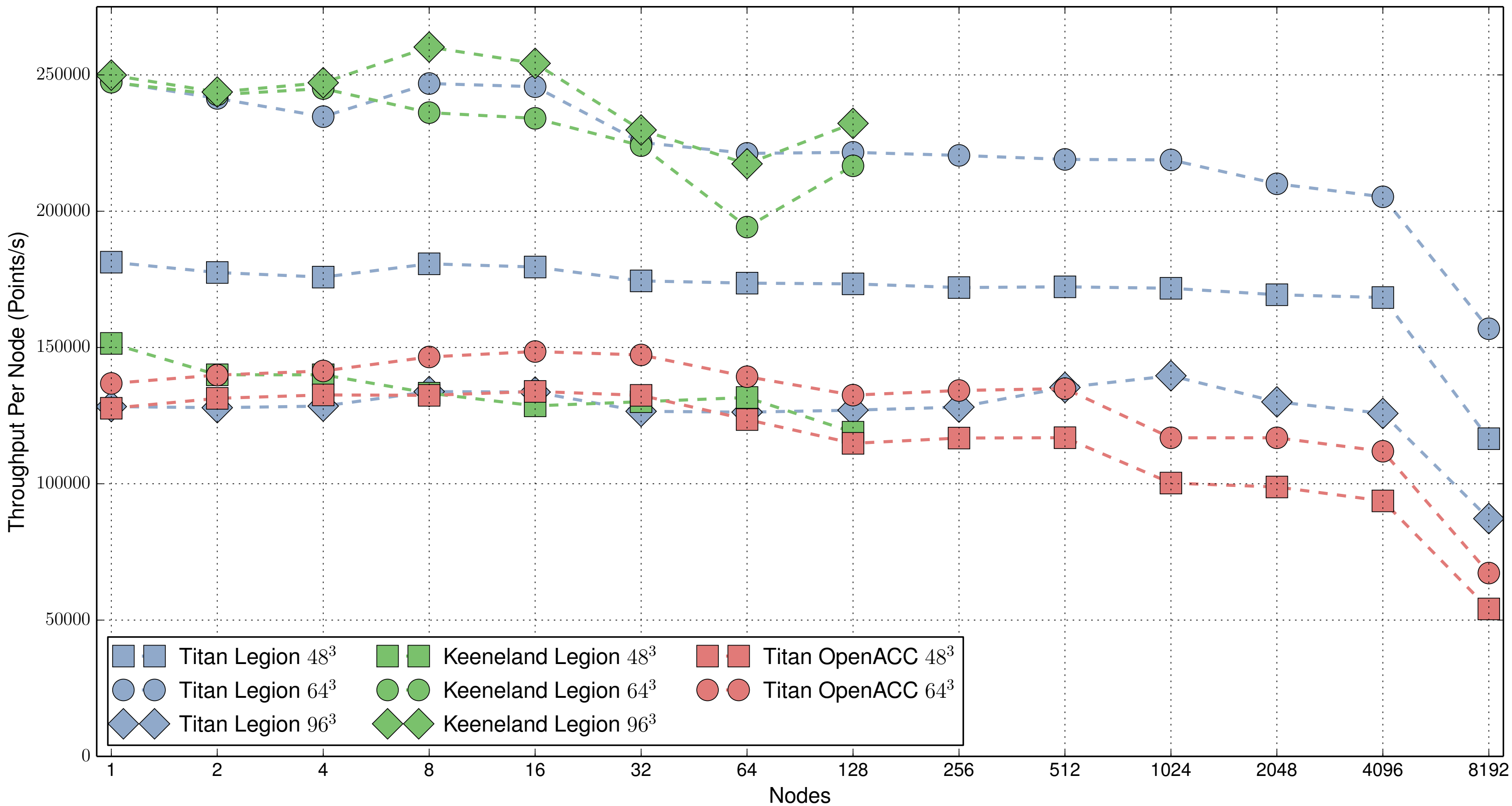
- Handle larger problem sizes per node
  - Higher computation-to-communication ratios
  - More power efficient
- Not enough room in 6 GB GPU framebuffer
  - OpenACC requires code changes
- Legion analysis is independent of problem size
  - Larger tasks -> fewer runtime cores



Performance Results

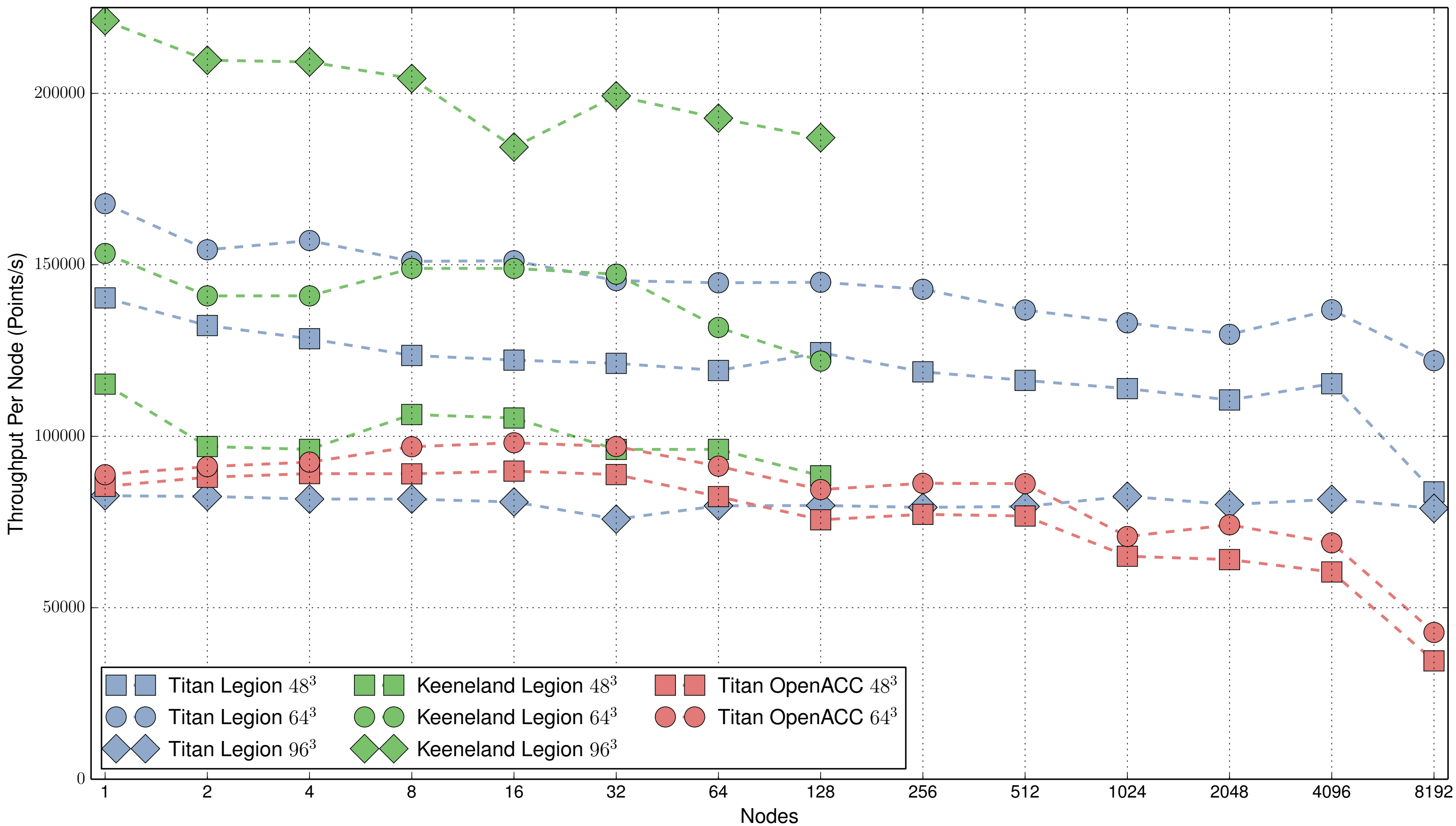
Legion S3D DME Performance

- **1.71X - 2.33X faster between 1024 and 8192 nodes**
- **Larger problem sizes have higher efficiency**



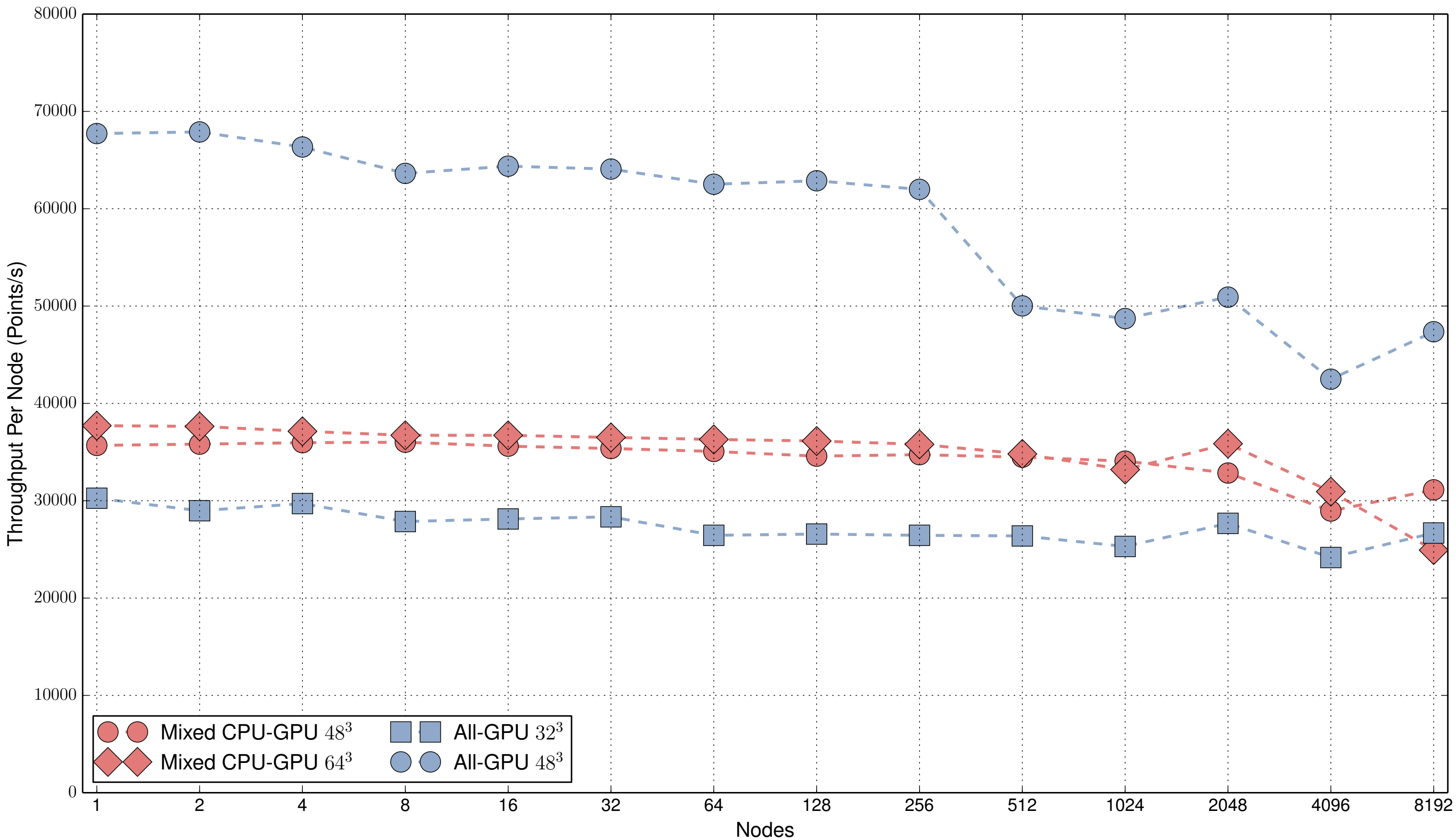
Legion Heptane Performance

- 1.73X - 2.85X faster between 1024 and 8192 nodes
- Higher throughput on Keeneland (balanced CPU+GPUs)



Legion PRF Performance

- 116 species mechanism, >2X as large as heptane
- Legion uses different mapping approach



Current Work

Liszt → Legion

```
local liszt InitLength
  var delta  =   e.head.pos
               - e.tail.pos
  e.rest_len = L.len(delta)
end
```

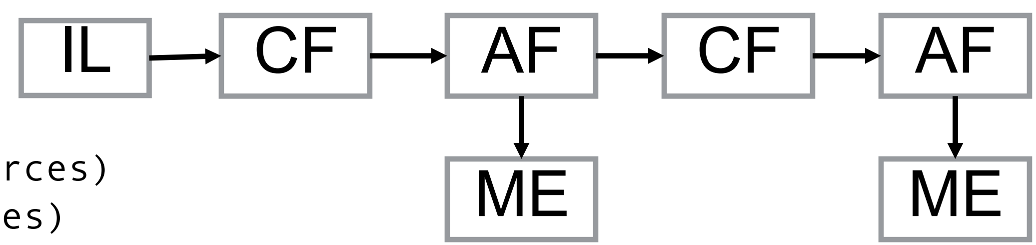
Phase Analysis

```
edges.head      : LEGION_READ
edges.tail      : LEGION_READ
edges.rest_len  : LEGION_READ_WRITE
```

Legion Permissions

```
dragon.edges:map(InitLength)

for i = 1,300 do
  dragon.vertices:foreach(ComputeForces)
  dragon.vertices:foreach(ApplyForces)
  dragon.vertices:foreach(MeasureEnergy)
end
```

Seq. Bulk Data-Parallel



Legion Task Graph

Legion

- Legion website: http://legion.stanford.edu
- Github repo: http://github.com/stanfordlegion
- Questions?