

Legion: Expressing Locality and Independence with Logical Regions

Authors: Bauer, M., Treichler, S., Slaughter, E., Aiken, A.
@International Conference on Supercomputing (SC 2012)

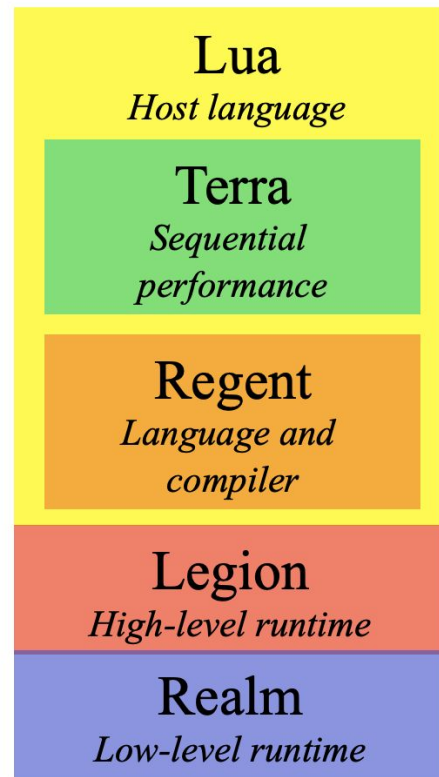
Presented by Alexey Voronin
For CSE598APK-FA18 @UIUC

Outline

- What is Legion
- Design Goals
- Programming Model
 - Data
 - Tasks
- Example
- Mapping Interface
- Run Time System
- Conclusion
- Afterword

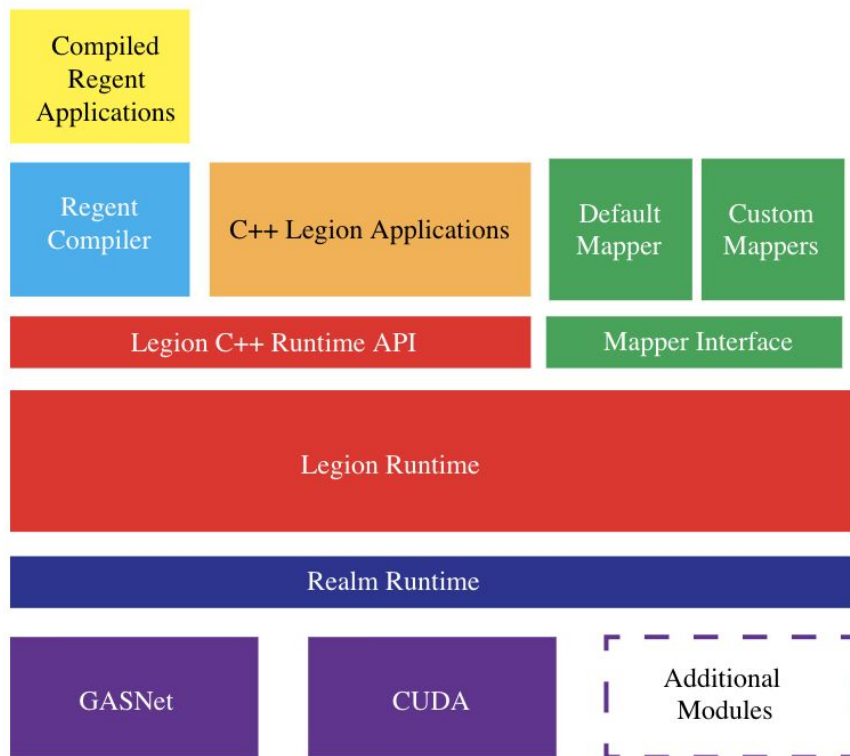
Legion & Regent

- Legion
 - A data-centric programming model
 - Asynchronous many task C++ runtime system
- Regent
 - Programming language for legion programming model
 - Current implementation is embedded in Lua
 - Has an optimizing compiler

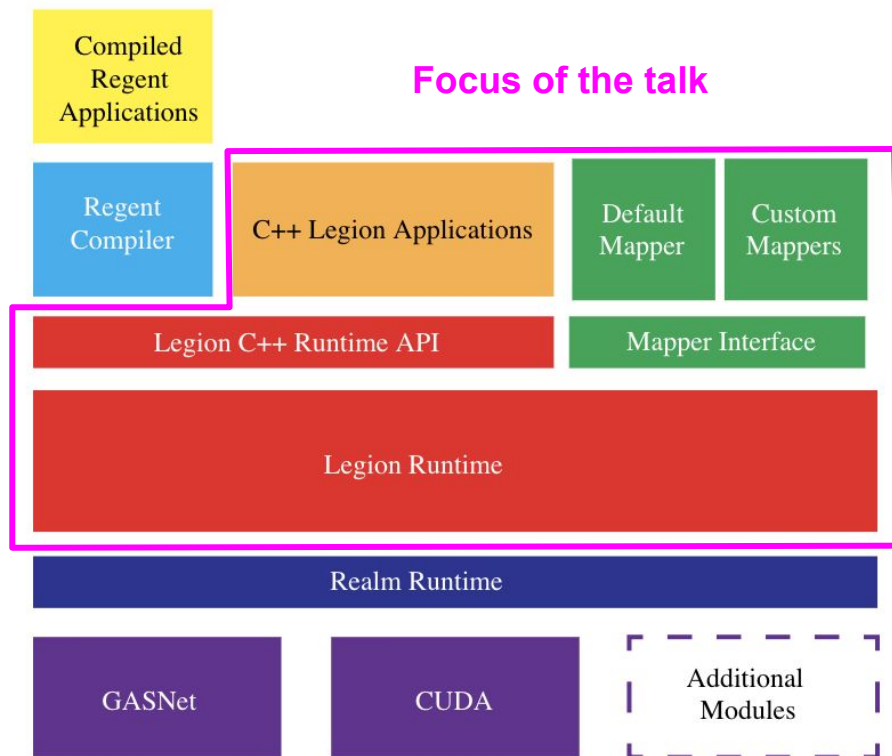


Regent Stack

System Architecture

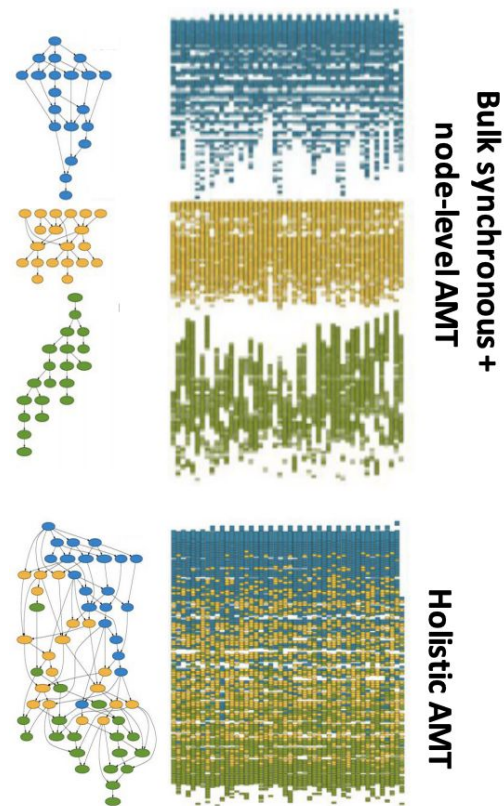


System Architecture




Legion/Regent Design Goals

- Programmability
 - Sequential Semantics
 - Readable code
 - Parallelism extracted automatically
- Throughput oriented
 - Latency of single task is irrelevant
 - Overall time is what matters
 - Good performance on heterogeneous architectures
- Run time decision making
 - Asynchronous execution
 - Runtime decision making because of software/hardware dynamics



Legion/Regent Design Goals

- Programmability
 - Sequential Semantics
 - Readable code
 - Parallelism extracted automatically
- Throughput oriented 
 - Latency of single task is irrelevant
 - Overall time is what matters
 - Good performance on heterogeneous architectures
- Run time decision making
 - Asynchronous execution
 - Runtime decision making because of software/hardware dynamics

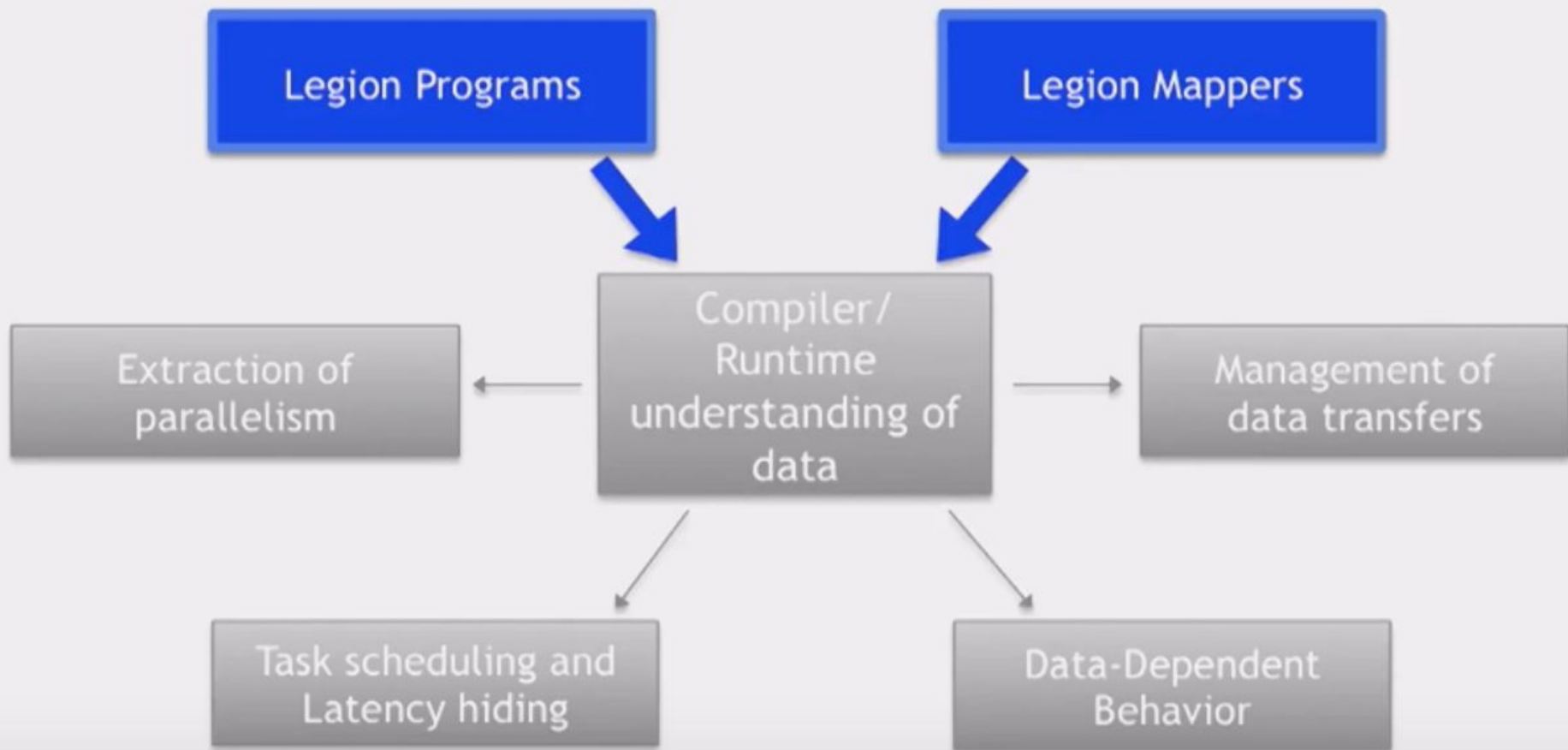
How?

Every core has

- queue of independent work
- queue of transfers to do

Minimize synchronization points

Separation of Concerns



High Level View of Legion

Tasks

(execution model)

Describe parallel execution elements and algorithmic operations with sequential semantics, out-of-order execution

Regions

(data model)

Describe decomposition of computational domain.

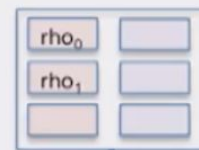
- Privileges (read-write, read-only, reduce)
- Coherence (exclusive, atomic)

Mapper

Describes how tasks and regions should be mapped to the target architecture

Mapper allows architecture-specific optimization without effecting the correctness of the task or domain descriptions

```
[=](int i) { rho(i) = ... }
```



*(pic source in comments)

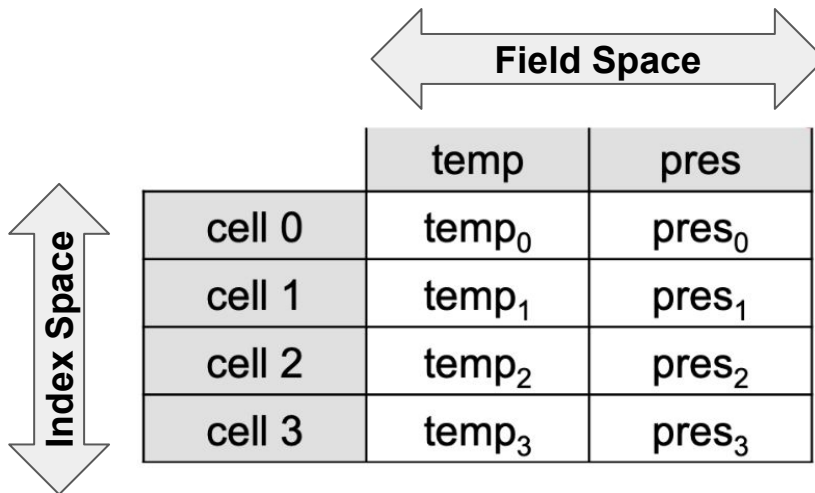
Logical Regions

- Typed Collection
 - Structured (like arrays)
 - Unstructured (like pointer data structures)

Suppose we want an array of cell_ts:

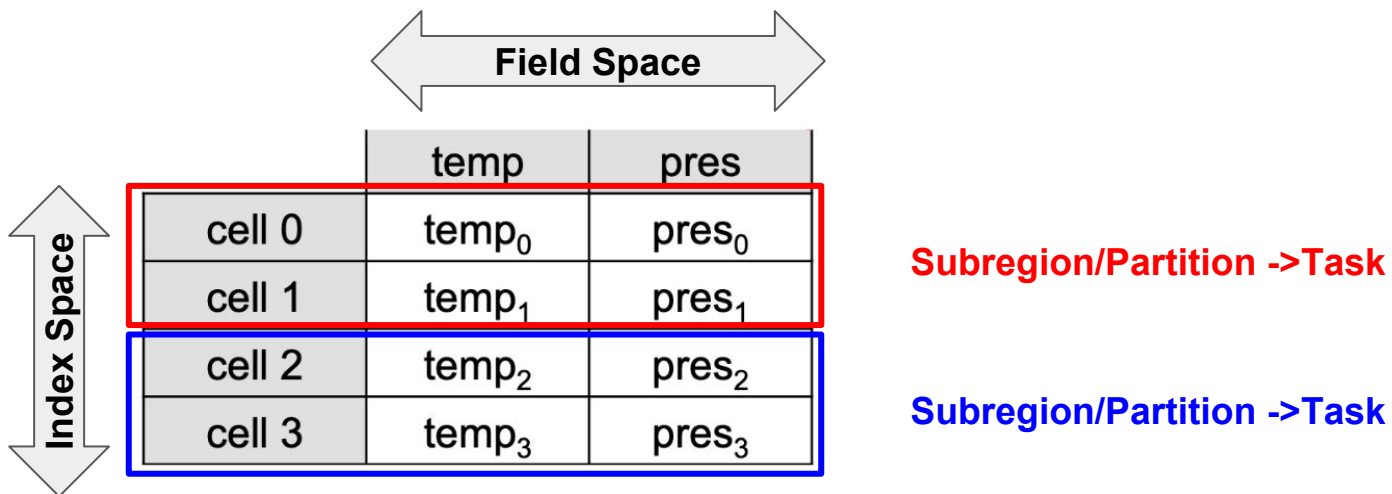
```
typedef struct cell_t {  
    double temp, pres;  
} cell_t;
```

In Legion it would look like this -



Structured Logical Regions

- Regions are split into partitions
 - Enables parallelism and allows tasks to run on each piece
 - Same data can be partitioned multiple (overlapping) ways

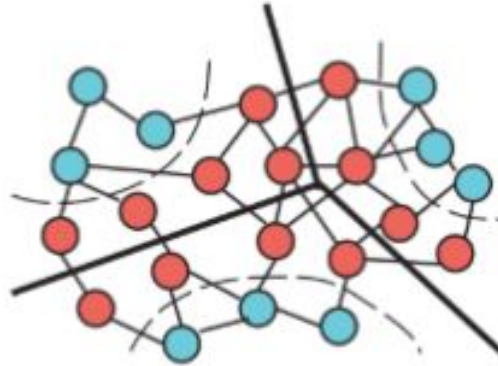


Tasks

- Units of parallel execution
- Runs until block or terminate
- Task that takes region arguments must declare what type of privileges it has in the region
 - Reads, writes, reduce
- Legion allows the user to define multiple different variants of the same task
 - CPU, GP, CPU that supports AVX instructions, etc.
 - Each variant is up to the user to implement
 - Mapper chooses which variant to use

Electrical Circuit Simulation Example

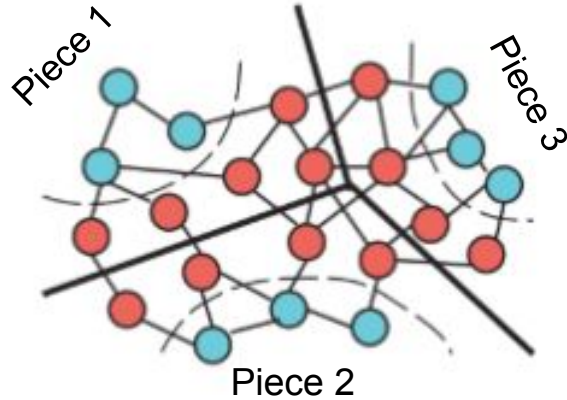
The circuit consists of a collection of **wires** and **nodes** where wires meet. At each time step the simulation calculates currents, distributes charges, and updates voltages.



How to group data into regions?

How are regions partitioned into subregions?

Electrical Circuit Simulation Example



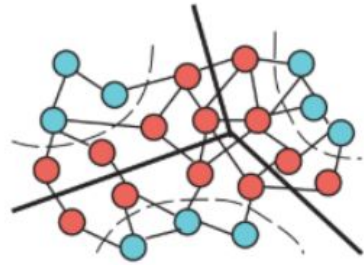
How to group data into regions?

How are regions partitioned into subregions?

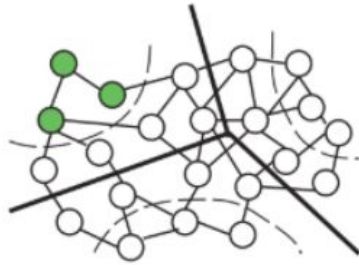
Regions:

- Nodes
 - Private
 - Shared
 - Ghost
- Wires

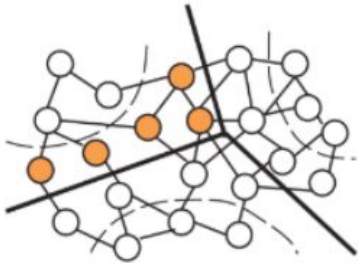
Circuit Example



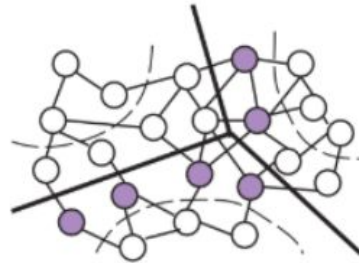
(b) p_nodes_pvs



(c) p_i

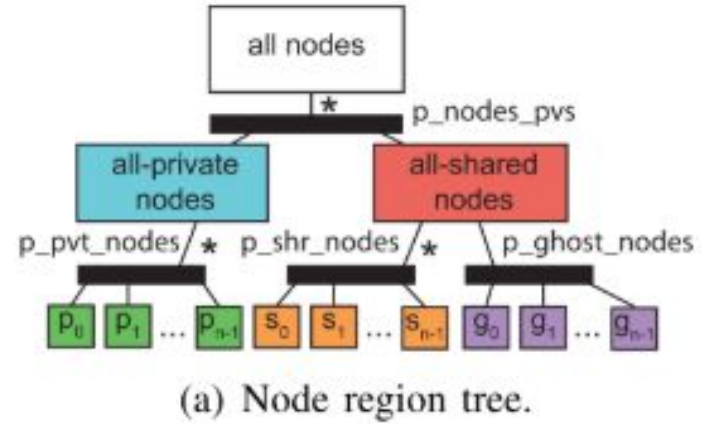


(d) s_i



(e) g_i

Fig. 1. Partitions of r_all_nodes .



(a) Node region tree.

This **region tree** data structure plays an important role in scheduling tasks for out-of-order execution

```

1 struct Node { float voltage, new_charge, capacitance; };
2 struct Wire<rn> { Node@rn in_node, out_node; float current, ... ; };
3 struct Circuit { region r_all_nodes; /* contains all nodes for the circuit */
4                 region r_all_wires; /* contains all circuit wires */ };
5 struct CircuitPiece {
6     region rn_pvt, rn_shr, rn_ghost; /* private, shared, ghost node regions */
7     region rw_pvt; /* private wires region */ };
8
9 void simulate_circuit(Circuit c, float dt) : RWE(c.r_all_nodes, c.r_all_wires)
10 {
11
12     // Partition of wires into MAX_PIECES pieces
13     partition<disjoint> p_wires = c.r_all_wires.partition(wire_owner_map);
14     // Partition nodes into two parts for all-private vs. all-shared
15     partition<disjoint> p_nodes_pvs = c.r_all_nodes.partition(node_sharing_map);
16
17     // Partition all-private into MAX_PIECES disjoint circuit pieces
18     partition<disjoint> p_pvt_nodes = p_nodes_pvs[0].partition(node_owner_map);
19     // Partition all-shared into MAX_PIECES disjoint circuit pieces
20     partition<disjoint> p_shr_nodes = p_nodes_pvs[1].partition(node_owner_map);
21     // Partition all-shared into MAX_PIECES ghost regions, which may be aliased
22     partition<aliased> p_ghost_nodes = p_nodes_pvs[1].partition(node_neighbor_map);
23
24     CircuitPiece pieces[MAX_PIECES];
25     for(i = 0; i < MAX_PIECES; i++)
26         pieces[i] = { rn_pvt: p_pvt_nodes[i], rn_shr: p_shr_nodes[i],
27                     rn_ghost: p_ghost_nodes[i], rw_pvt: p_wires[i] };
28     for (t = 0; t < TIME_STEPS; t++) {
29         spawn (i = 0; i < MAX_PIECES; i++) calc_new_currents(pieces[i]);
30         spawn (i = 0; i < MAX_PIECES; i++) distribute_charge(pieces[i], dt);
31         spawn (i = 0; i < MAX_PIECES; i++) update_voltages(pieces[i]);
32     }
33 }
34
35 }
36

```

Specifies that the regions are accessed with read-write **privileges** and exclusive **coherence** (i.e., no other task can access these two regions concurrently).


```

1 struct Node { float voltage, new_charge, capacitance; };
2 struct Wire<rn> { Node@rn in_node, out_node; float current, ... ; };
3 struct Circuit { region r_all_nodes; /* contains all nodes for the circuit */
4                 region r_all_wires; /* contains all circuit wires */ };
5 struct CircuitPiece {
6     region rn_pvt, rn_shr, rn_ghost; /* private, shared, ghost node regions */
7     region rw_pvt; /* private wires region */ };
8
9 void simulate_circuit(Circuit c, float dt) : RWE(c.r_all_nodes, c.r_all_wires)
10 {
11     // Partition of wires into MAX_PIECES pieces
12     partition<disjoint> p_wires = c.r_all_wires.partition(wire_owner_map);
13     // Partition nodes into two parts for all-private vs. all-shared
14     partition<disjoint> p_nodes_pvs = c.r_all_nodes.partition(node_sharing_map);
15
16     // Partition all-private into MAX_PIECES disjoint circuit pieces
17     partition<disjoint> p_pvt_nodes = p_nodes_pvs[0].partition(node_owner_map);
18     // Partition all-shared into MAX_PIECES disjoint circuit pieces
19     partition<disjoint> p_shr_nodes = p_nodes_pvs[1].partition(node_owner_map);
20     // Partition all-shared into MAX_PIECES ghost regions, which may be aliased
21     partition<aliased> p_ghost_nodes = p_nodes_pvs[1].partition(node_neighbor_map);
22
23     CircuitPiece pieces[MAX_PIECES];
24     for(i = 0; i < MAX_PIECES; i++)
25         pieces[i] = { rn_pvt: p_pvt_nodes[i], rn_shr: p_shr_nodes[i],
26                     rn_ghost: p_ghost_nodes[i], rw_pvt: p_wires[i] };
27     for (t = 0; t < TIME_STEPS; t++) {
28         spawn (i = 0; i < MAX_PIECES; i++) calc_new_currents(pieces[i]);
29         spawn (i = 0; i < MAX_PIECES; i++) distribute_charge(pieces[i], dt);
30         spawn (i = 0; i < MAX_PIECES; i++) update_voltages(pieces[i]);
31     }
32 }
33
34 }
35
36 }

```

Specifies that the regions are accessed with read-write privileges and exclusive coherence (i.e., no other task can access these two regions concurrently).

```

1 struct Node { float voltage, new_charge, capacitance; };
2 struct Wire<rn> { Node@rn in_node, out_node; float current, ... ; };
3 struct Circuit { region r_all_nodes; /* contains all nodes for the circuit */
4                 region r_all_wires; /* contains all circuit wires */ };
5 struct CircuitPiece {
6     region rn_pvt, rn_shr, rn_ghost; /* private, shared, ghost node regions */
7     region rw_pvt; /* private wires region */ };
8
9 void simulate_circuit(Circuit c, float dt) : RWE(c.r_all_nodes, c.r_all_wires)
10 {
11     // Partition of wires into MAX_PIECES pieces
12     partition<disjoint> p_wires = c.r_all_wires.partition(wire_owner_map);
13     // Partition nodes into two parts for all-private vs. all-shared
14     partition<disjoint> p_nodes_pvs = c.r_all_nodes.partition(node_sharing_map);
15
16     // Partition all-private into MAX_PIECES disjoint circuit pieces
17     partition<disjoint> p_pvt_nodes = p_nodes_pvs[0].partition(node_owner_map);
18     // Partition all-shared into MAX_PIECES disjoint circuit pieces
19     partition<disjoint> p_shr_nodes = p_nodes_pvs[1].partition(node_owner_map);
20     // Partition all-shared into MAX_PIECES ghost regions, which may be aliased
21     partition<aliased> p_ghost_nodes = p_nodes_pvs[1].partition(node_neighbor_map);
22
23     CircuitPiece pieces[MAX_PIECES];
24     for(i = 0; i < MAX_PIECES; i++)
25         pieces[i] = { rn_pvt: p_pvt_nodes[i], rn_shr: p_shr_nodes[i],
26                     rn_ghost: p_ghost_nodes[i], rw_pvt: p_wires[i] };
27
28     for (t = 0; t < TIME_STEPS; t++) {
29         spawn (i = 0; i < MAX_PIECES; i++) calc_new_currents(pieces[i]);
30         spawn (i = 0; i < MAX_PIECES; i++) distribute_charge(pieces[i], dt);
31         spawn (i = 0; i < MAX_PIECES; i++) update_voltages(pieces[i]);
32     }
33 }
34
35 }
36

```

Specifies that the regions are accessed with read-write privileges and exclusive coherence (i.e., no other task can access these two regions concurrently).

spawn indicates a task call.
→ parallelism

interpass dependencies are determined automatically based on the region access declarations.

Circuit Example (cont.)

```
37         // ROE = Read-Only-Exclusive
38 void calc_new_currents(CircuitPiece piece):
39     RWE(piece.rw_pvt), ROE(piece.rn_pvt, piece.rn_shr, piece.rn_ghost) {
40     foreach(w : piece.rw_pvt)
41         w->current = (w->in_node->voltage - w->out_node->voltage) / w->resistance;
42     }
43         // RdA = Reduce-Atomic
44 void distribute_charge(CircuitPiece piece, float dt):
45     ROE(piece.rw_pvt), RdA(piece.rn_pvt, piece.rn_shr, piece.rn_ghost) {
46     foreach(w : piece.rw_pvt) {
47         w->in_node->new_charge += -dt * w->current;
48         w->out_node->new_charge += dt * w->current;
49     }
50 }
51
52 void update_voltages(CircuitPiece piece): RWE(piece.rn_pvt, piece.rn_shr) {
53     foreach(n : piece.rn_pvt, piece.rn_shr) {
54         n->voltage += n->new_charge / n->capacitance;
55         n->new_charge = 0;
56     }
57 }
```

Circuit Example (cont.)

```
37         // ROE = Read-Only-Exclusive
38 void calc_new_currents(CircuitPiece piece):
39     RWE(piece.rw_pvt), ROE(piece.rn_pvt, piece.rn_shr, piece.rn_ghost) {
40     foreach(w : piece.rw_pvt)
41         w->current = (w->in_node->voltage - w->out_node->voltage) / w->resistance;
42     }
43         // RdA = Reduce-Atomic
44 void distribute_charge(CircuitPiece piece, float dt):
45     ROE(piece.rw_pvt), RdA(piece.rn_pvt, piece.rn_shr, piece.rn_ghost) {
46     foreach(w : piece.rw_pvt) {
47         w->in_node->new_charge += -dt * w->current;
48         w->out_node->new_charge += dt * w->current;
49     }
50 }
51
52 void update_voltages(CircuitPiece piece): RWE(piece.rn_pvt, piece.rn_shr) {
53     foreach(n : piece.rn_pvt, piece.rn_shr) {
54         n->voltage += n->new_charge / n->capacitance;
55         n->new_charge = 0;
56     }
57 }
```

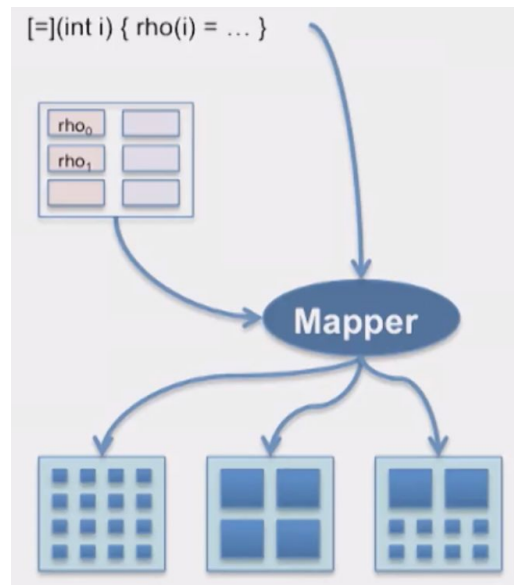
Legion uses tasks' region arguments to compute which tasks can run in parallel

Mapping Interface

- Isolates mapping decisions from application code
- Gives programmers control over where tasks run and where region instances are placed
- Allows for dynamic decision making based on input data
- Currently done at Legion level

Properties:

- Program correctness is unaffected by mapper decisions
- Isolates machine-specific decisions to the mapper, resulting in portability of Legion programs

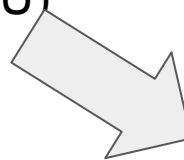


Mapping Interface

Consists of 10 methods that Legion runtime system call for mapping decisions.

A mapper implementing these methods has access to following properties

- List of processors and their type (e.g. CPU, GPU)
- List of memories visible to each processor
- Related latencies and bandwidths



- Processors
 - LOC
 - TOC
 - PROC_SET
 - UTILITY
 - IO
- Memories
 - GLOBAL
 - SYSTEM
 - RDMA
 - FRAME_BUFFER
 - ZERO_COPY
 - DISK
 - HDF5

Three most important interface calls:

- `select_initial_processor`
- `permit_task_steal`
- `map_task_region`

Mapping Interface

Three most important interface calls:

- `select_initial_processor`
 - SOOP will ask for a processor for a task t
 - Mapper can keep task t local or send it to any other processor
- `permit_task_steal`
 - SOOP asks which tasks can be stolen
 - If no stealing allowed, return empty set
- `map_task_region`
 - For each logical region r used by a task, a SOOP asks for a prioritized list of memories where a physical instance of r should be placed
 - Mapper returns a priority list of memories in which the SOOP should attempt to either reuse or create a physical instance of r

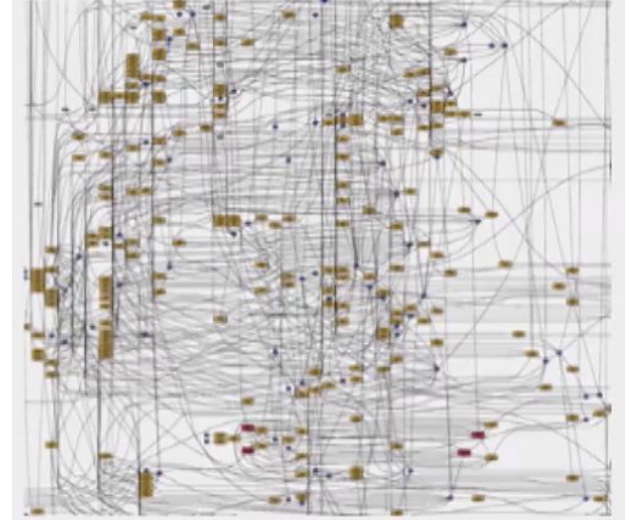
Default Mapping Interface

Legion provides a default mapping interface for a quick start:

- `select_initial_processor`
 - Mapper checks the type of processors for which task `t` has implementations
 - If the fastest implementation is for the local processor, the mapper keeps `t` local
- `permit_task_steal`
 - Mapper inspects the logical regions for the task being stolen and marks that other tasks using the same logical regions should be stolen as well
- `map_task_region`
 - Select the final (set of) processor(s) that the task will be executed on
 - Select the variant of the task to execute
 - Select the physical instances to hold the data for each logical region
 - Optionally select the task priority

Software Out of Order Processor (SOOP)

- Dynamically schedules a stream of tasks
- Constrained by region dependences
- Pipelined, distributed, and extracts nested parallelism from subtasks
- Must hide extremely long latencies
 - Deferred execution model
 - Tasks run until they block or terminate
 - Blocking does not prevent independent work from being done
 - Blocking does prevent the task from continuing and launching more tasks



Control flow graph of one step on one node of a mini app *(pic source in comments)

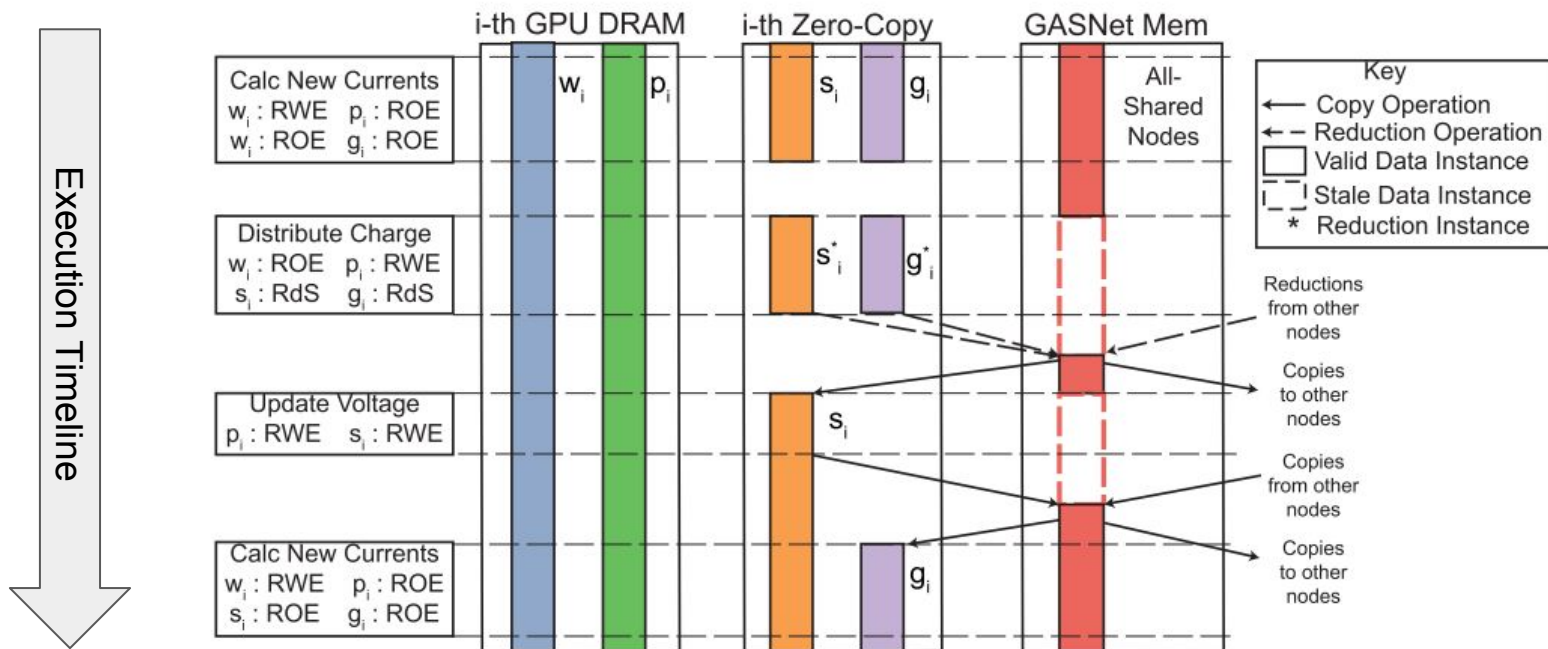
More on (Non-)Blocking Execution

- Futures
 - Objects which represent a pending return value from a task
 - Two ways to use them,
 - Blocking
 - Task pauses until the subtask that is completing the future returns (bad for performance)
 - Non-blocking
 - Can pass it as an argument to a function, which won't execute until the result becomes available
 - Allows the Legion runtime to discover as much task-level parallelism as possible.

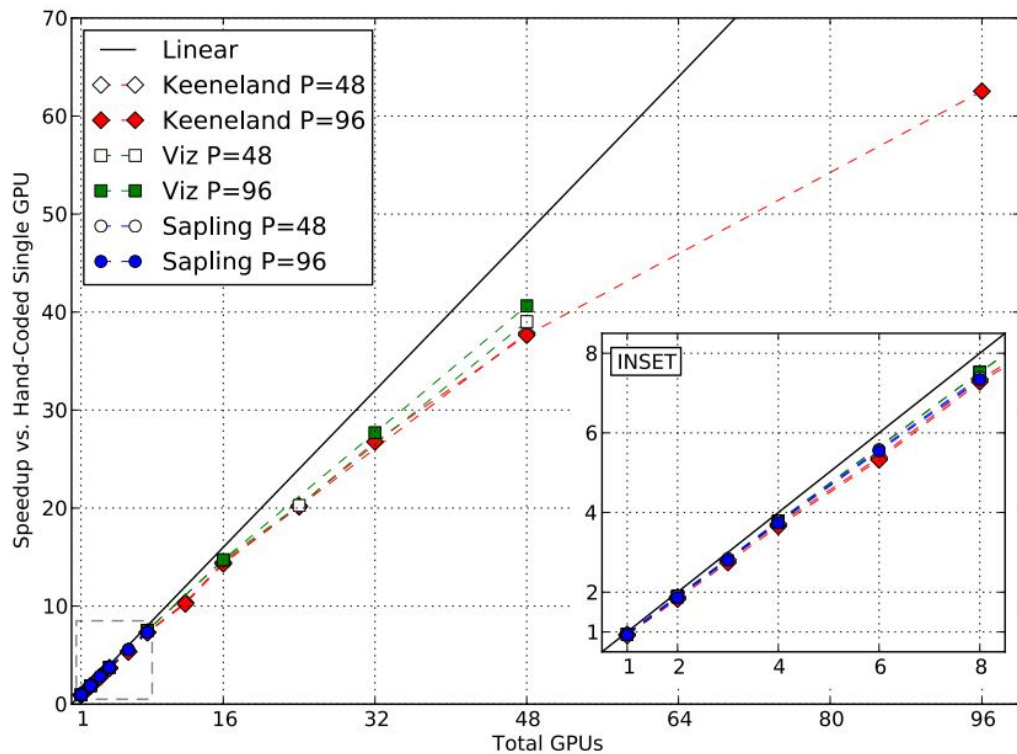
Circuit Simulation

- Machines
 - Linux based
 - Pthreads for managing CPU threads
 - CUDA for GPUs
 - GASNet for inter-node communication
- Legion Set-up
 - Runtime handles all of the resource allocation, scheduling, and data movement across the cluster of GPUs.
 - Mapper queries the list of GPUs in the machine and identifies each GPU's framebuffer memory and zero-copy memory
 - Circuit partitions are assigned a home GPU in round-robin fashion

Tasks and data for the circuit simulation on a cluster of GPUs

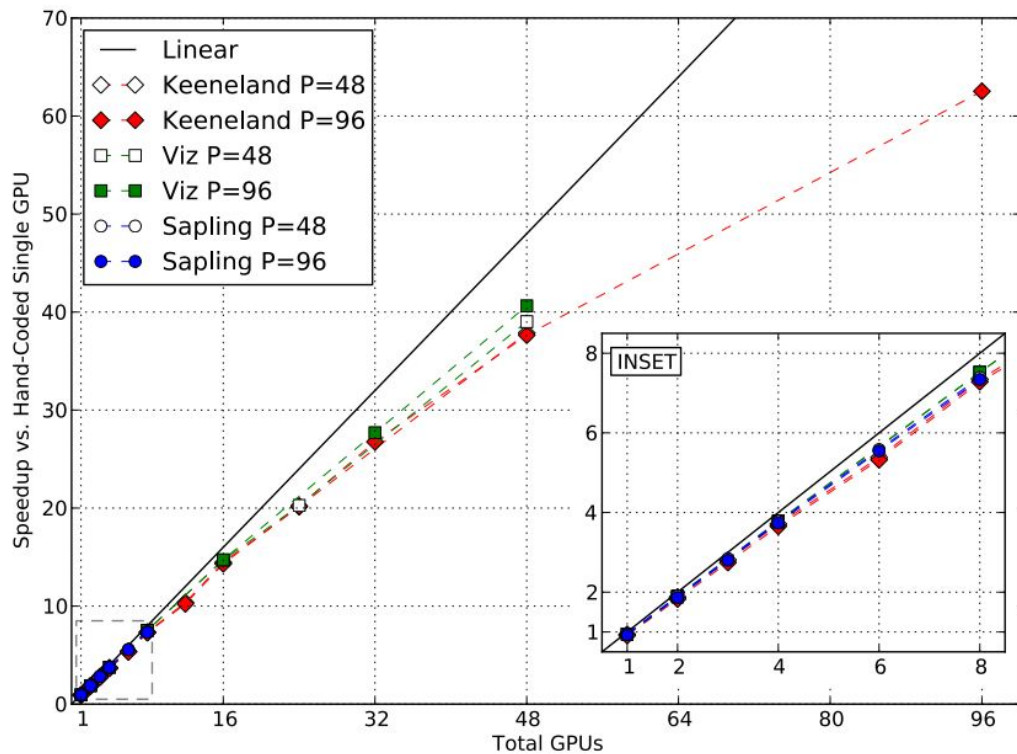


Circuit simulation speed relative to single-GPU implementation

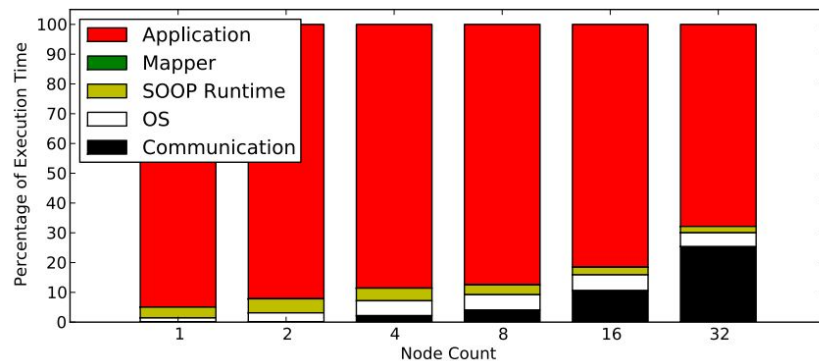


Cluster	Sapling	Viz	Keeneland
Nodes	4	10	32 (120)
CPUs/Node	2x Xeon 5680	2x Xeon 5680	2x Xeon 5660
HyperThreading	on	off	off
GPUs/Node	2x Tesla C2070	5x Quadro Q5000	3x Tesla M2090
DRAM/Node	48 GB	24 GB	24 GB
Infiniband	2x QDR	QDR	2x QDR

Circuit simulation speed relative to single-GPU implementation



Cluster	Sapling	Viz	Keeneland
Nodes	4	10	32 (120)
CPUs/Node	2x Xeon 5680	2x Xeon 5680	2x Xeon 5660
HyperThreading	on	off	off
GPUs/Node	2x Tesla C2070	5x Quadro Q5000	3x Tesla M2090
DRAM/Node	48 GB	24 GB	24 GB
Infiniband	2x QDR	QDR	2x QDR



(b) Overhead of circuit simulation on Keeneland with 3 GPUs/node.

Applications Using Legion

- Snap
 - Neutral Particle Transport mini-app
 - Results
 - Different mappers allows the application to specialize itself for different target architectures
 - Verbose codebase due to many tasks variants
 - Legion specific calls amount to only in 2% of overall code
- Legion Version of High Performance Conjugate Gradients (HPCG) Benchmark
 - No performance results published
- Lux
 - A distributed multi-GPU system for fast graph processing.
 - Much better multi-GPU performance than competitors, mostly due to smarter load balancing

Conclusion

Legion Goals

- High Performance
- Performance Portability
- Programmability (using sequential semantics)

Central role of Legion

- Schedule tasks in a way that preserves “locality” and “independence”
- Determine when to run the tasks

What legion doesn't do

- Automatically generate tasks
- Automatically map tasks/data to hardware

Afterword

- Don't spend too much on the paper
- Go to <https://legion.stanford.edu>
 - Tutorials
 - Rudimentary
 - Bootcamp
 - 20+ hours of 'getting started' videos
- [Regent](#)
- [ATPESC training video](#)
- Asynchronous Many-Task Runtime System Analysis and Assessment for Next Generation Platforms [Report](#) (Sandia, 2015)
- Copper Mountain Conference on Multigrid Methods [Presentation](#) (2015)