Legion Overview

Alex Aiken

December 4, 2014

http://legion.stanford.edu

Logistics

- Wireless
    - Choose “Stanford Visitor” network, follow directions
    - Bootcamp slides @ legion.stanford.edu
- Thursday
    - Extending the schedule by 15 minutes
    - Parking
    - Lunch
    - Dinner
- Friday
    - Different building: Gates 505

Team

- Alex Aiken
- Mike Bauer (Nvidia)
- Zhihao Jia
- Wonchan Lee
- Elliott Slaughter
- Sean Treichler

- Charles Ferenbaugh
- Sam Gutierrez
- Pat McCormick

Legion

- A programming model for heterogeneous, distributed machines

- Heterogeneous
  - Mixed CPUs and GPUs

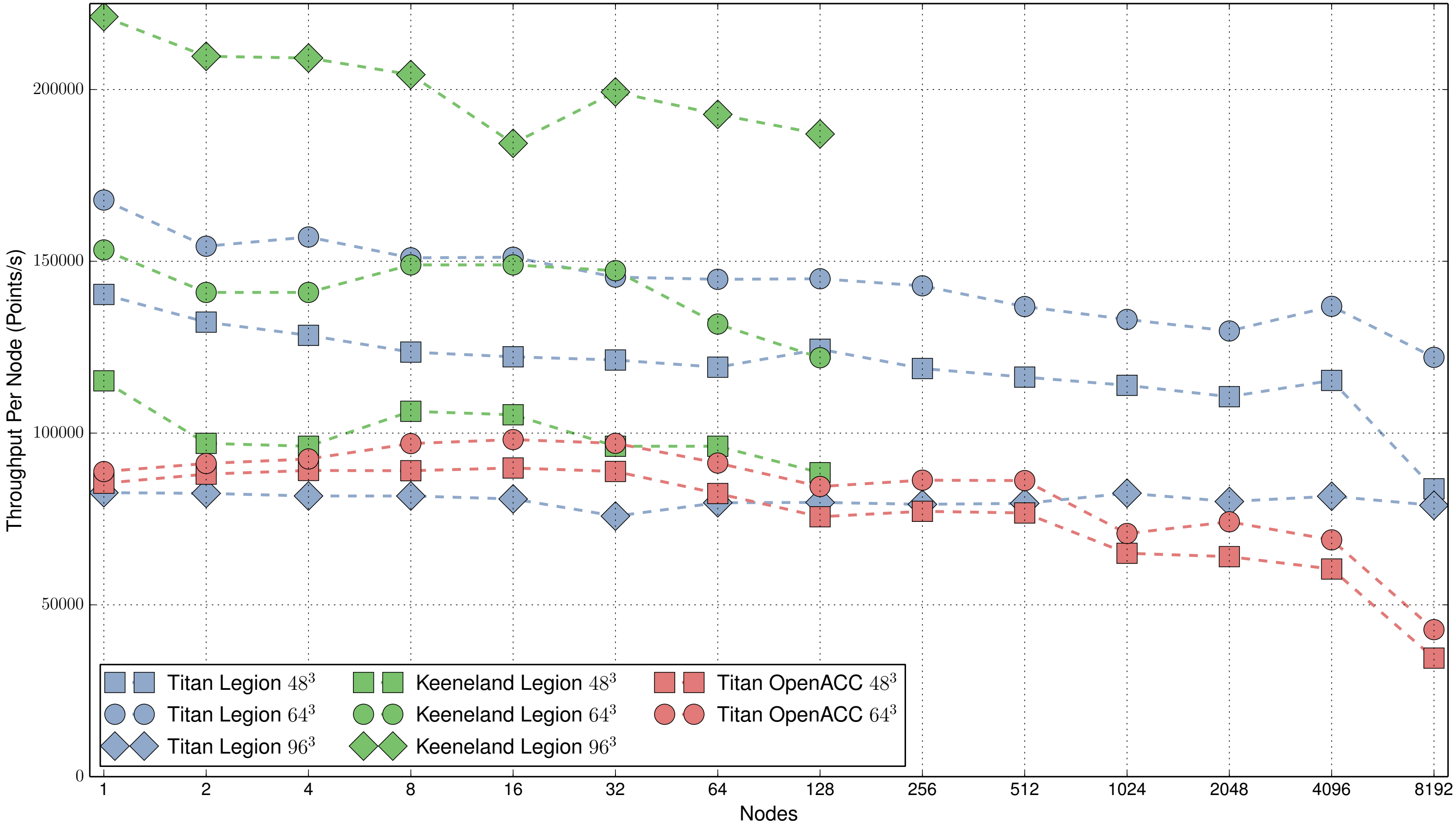
- Distributed
  - Large spread, and variability, of communication latencies
  - Caches, RAM, NUMA, network, …

One Slide History

- Started in 2011
- First version in 2012
- S3D implementation in 2013
  - Collaboration with Jackie Chen's group at Sandia
  - Part of the ExaCT Center
  - Drove many feature changes/additions
  - And many optimizations/improvements
- Emphasis on scaling up in 2014
  - S3D on 8,000 Titan nodes

Legion S3D Heptane Performance

- **1.73X - 2.85X faster between 1024 and 8192 nodes**



December 4, 2014

http://legion.stanford.edu

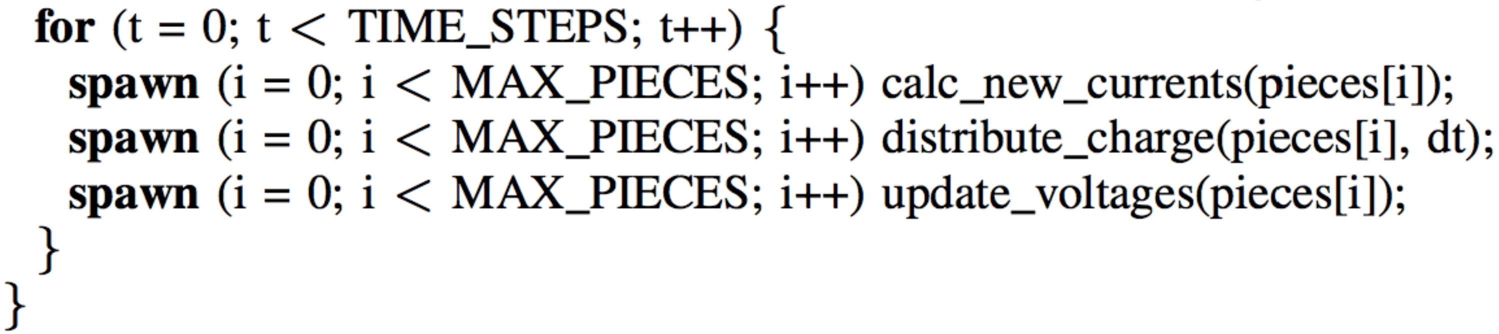
Bootcamp Focus

- Writing Legion programs
  - Different from the academic papers
  - Cover many pragmatic, usability aspects
- This morning: The programming model
  - Tasks, regions, mapping
- This afternoon: Everything else
  - Structuring applications
  - Debugging & profiling
- Tomorrow: Working with application groups

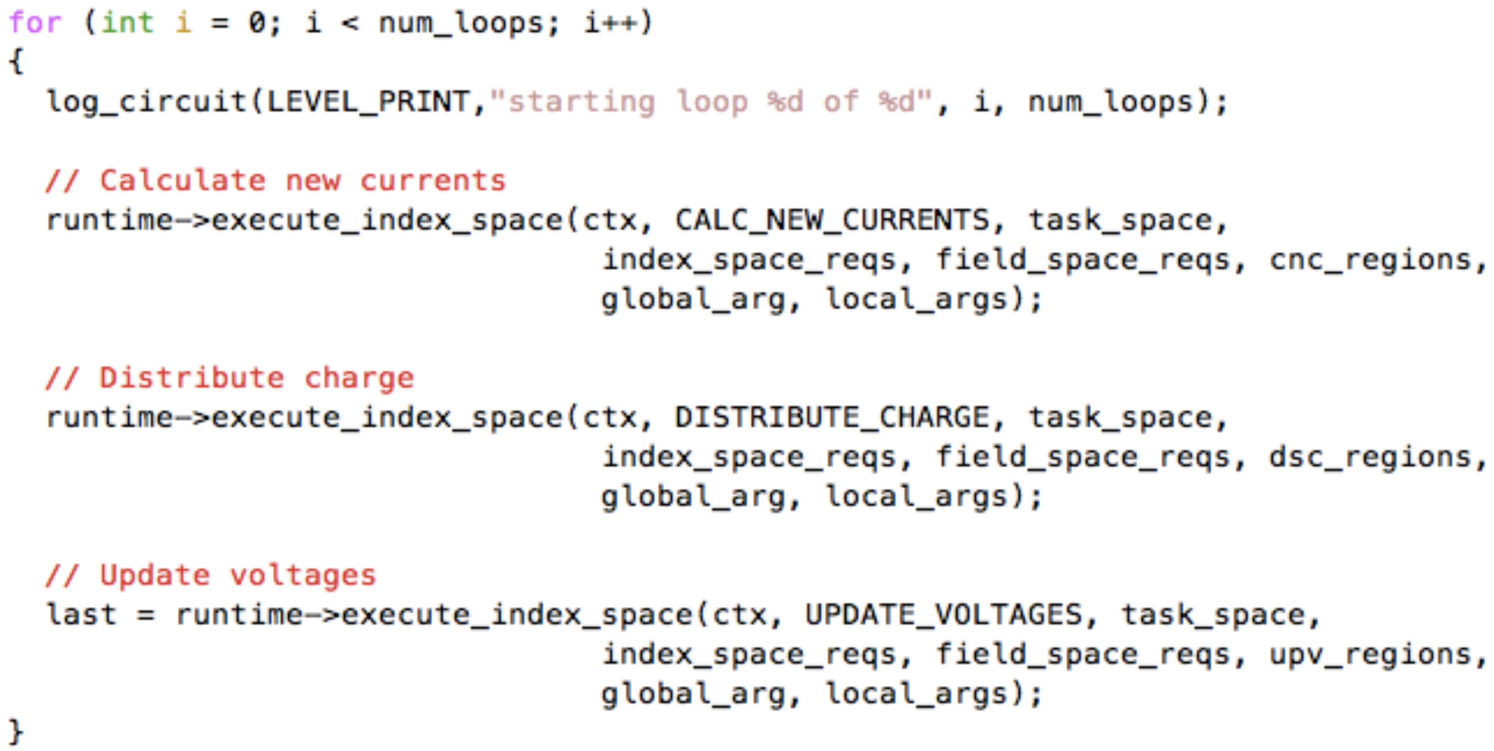
Philosophy

- Designed to be a real programming system
- Good abstractions, clear semantics
- But can also “open the hood”
  - Ways to drop down to lower-levels of abstraction
  - Within the programming model

Example Code

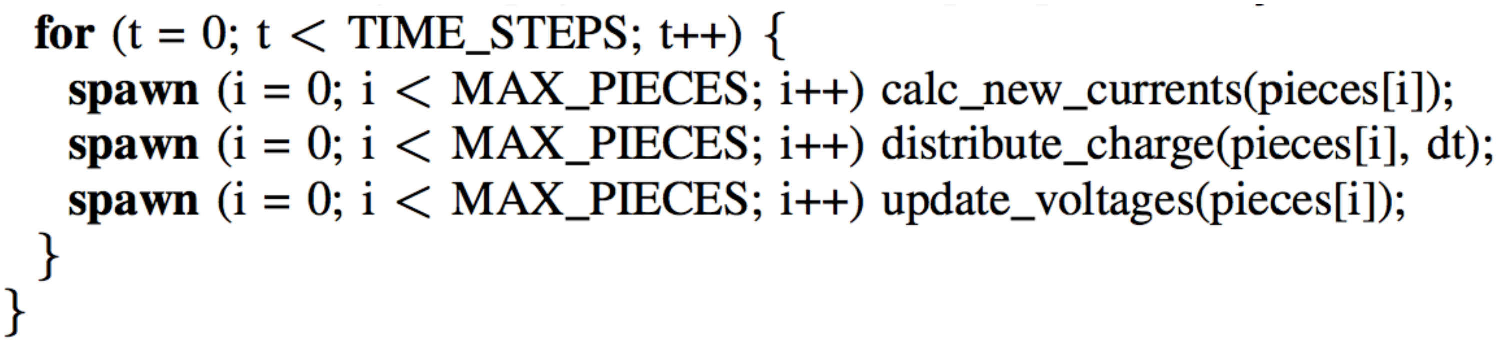


```
  for (t = 0; t < TIME_STEPS; t++) {
    spawn (i = 0; i < MAX_PIECES; i++) calc_new_currents(pieces[i]);
    spawn (i = 0; i < MAX_PIECES; i++) distribute_charge(pieces[i], dt);
    spawn (i = 0; i < MAX_PIECES; i++) update_voltages(pieces[i]);
  }
}
```



```
for (int i = 0; i < num_loops; i++)
{
  log_circuit(LEVEL_PRINT,"starting loop %d of %d", i, num_loops);

  // Calculate new currents
  runtime->execute_index_space(ctx, CALC_NEW_CURRENTS, task_space,
                                index_space_reqs, field_space_reqs, cnc_regions,
                                global_arg, local_args);

  // Distribute charge
  runtime->execute_index_space(ctx, DISTRIBUTE_CHARGE, task_space,
                                index_space_reqs, field_space_reqs, dsc_regions,
                                global_arg, local_args);

  // Update voltages
  last = runtime->execute_index_space(ctx, UPDATE_VOLTAGES, task_space,
                                index_space_reqs, field_space_reqs, upv_regions,
                                global_arg, local_args);
}
```

First Point



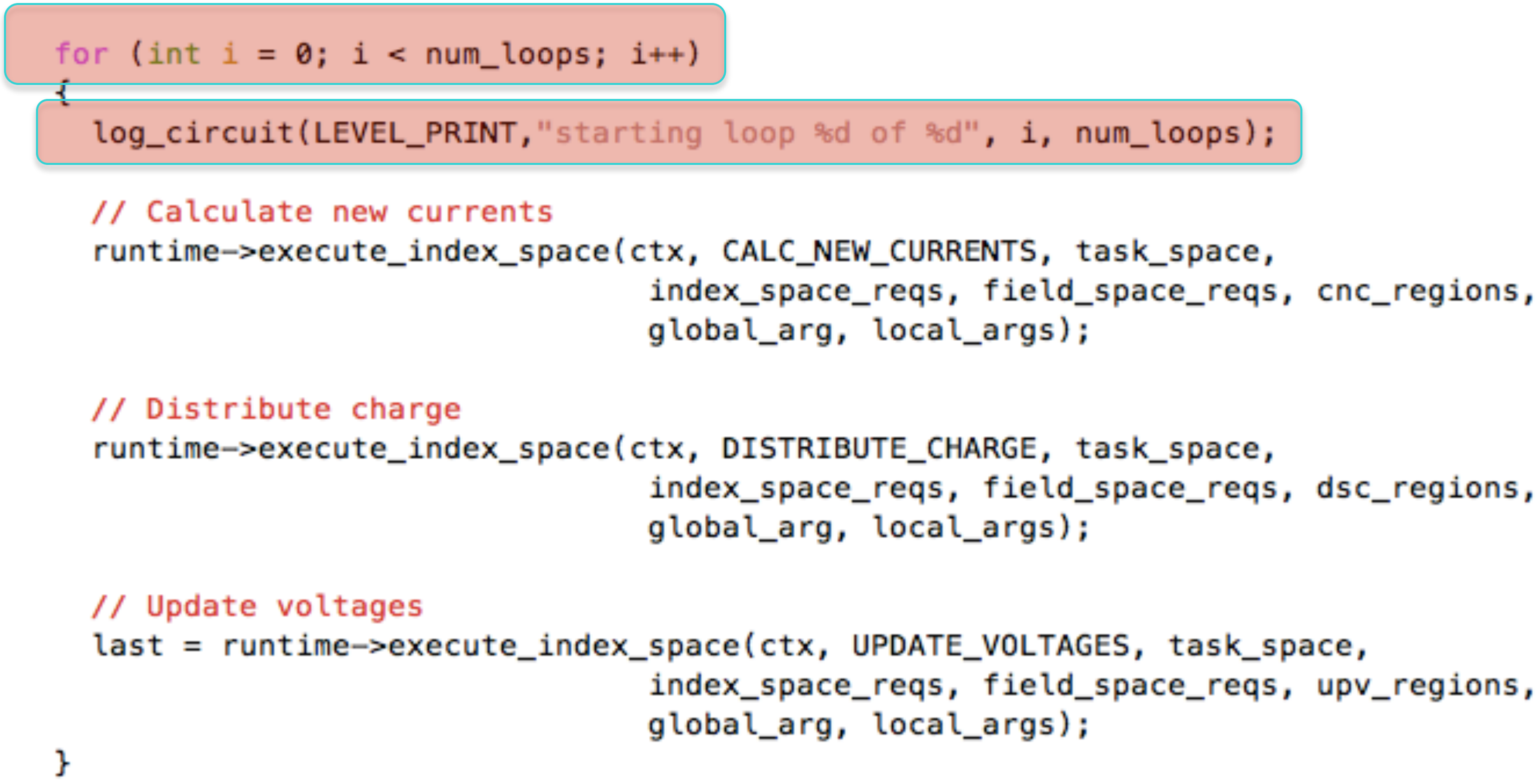
```
for (t = 0; t < TIME_STEPS; t++) {
  spawn (i = 0; i < MAX_PIECES; i++) calc_new_currents(pieces[i]);
  spawn (i = 0; i < MAX_PIECES; i++) distribute_charge(pieces[i], dt);
  spawn (i = 0; i < MAX_PIECES; i++) update_voltages(pieces[i]);
 }
}
```

- **Legion has a *sequential* semantics**
  - Easy to reason about
  - But see discussion of advanced features this afternoon

- **Not like**
  - **MPI**
  - **OpenACC**
  - **CUDA**

Second Point

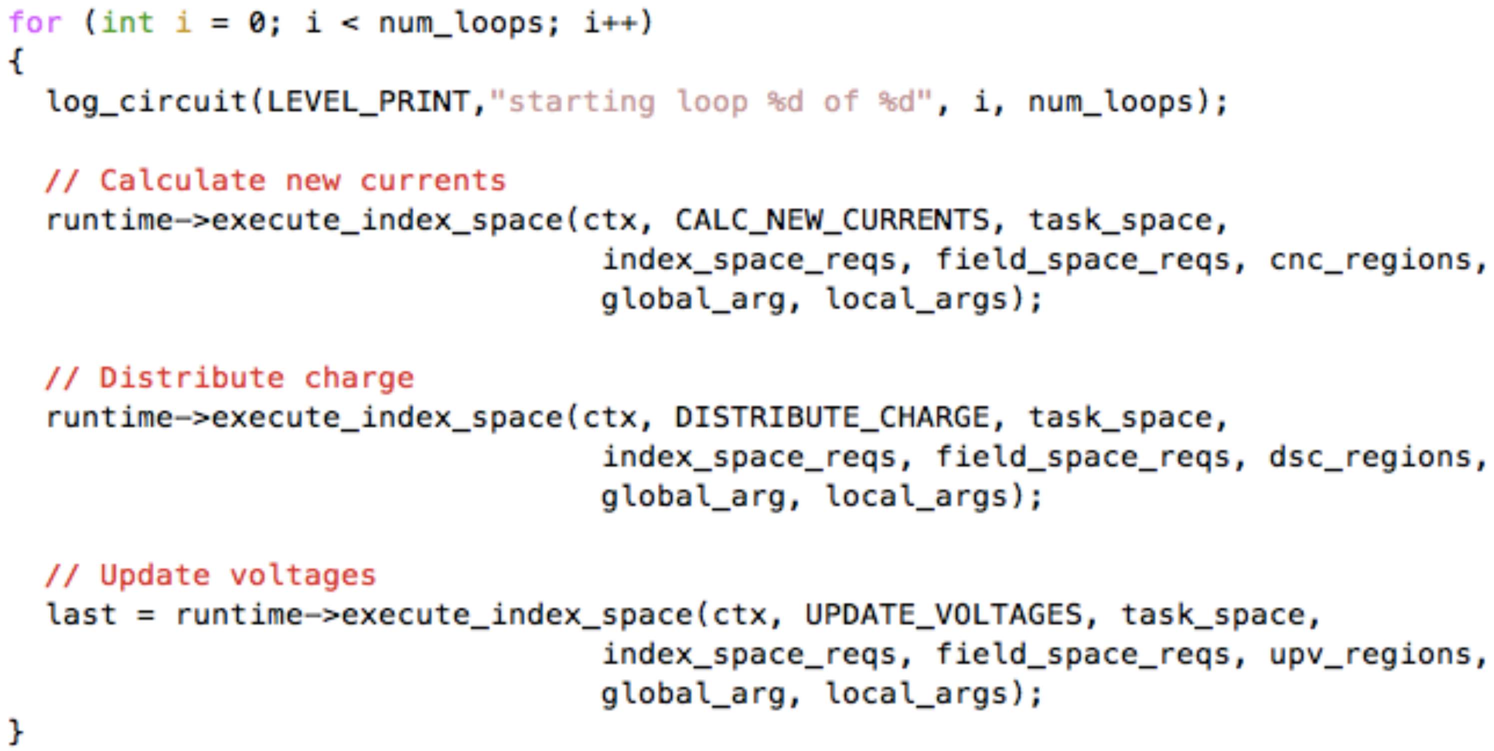
- A programming *model*
  - embedded in C++
  - but see discussion of future Legion compiler later today



```
for (int i = 0; i < num_loops; i++)
{
  log_circuit(LEVEL_PRINT,"starting loop %d of %d", i, num_loops);

  // Calculate new currents
  runtime->execute_index_space(ctx, CALC_NEW_CURRENTS, task_space,
                               index_space_reqs, field_space_reqs, cnc_regions,
                               global_arg, local_args);

  // Distribute charge
  runtime->execute_index_space(ctx, DISTRIBUTE_CHARGE, task_space,
                               index_space_reqs, field_space_reqs, dsc_regions,
                               global_arg, local_args);

  // Update voltages
  last = runtime->execute_index_space(ctx, UPDATE_VOLTAGES, task_space,
                               index_space_reqs, field_space_reqs, upv_regions,
                               global_arg, local_args);
}
```

Third Point

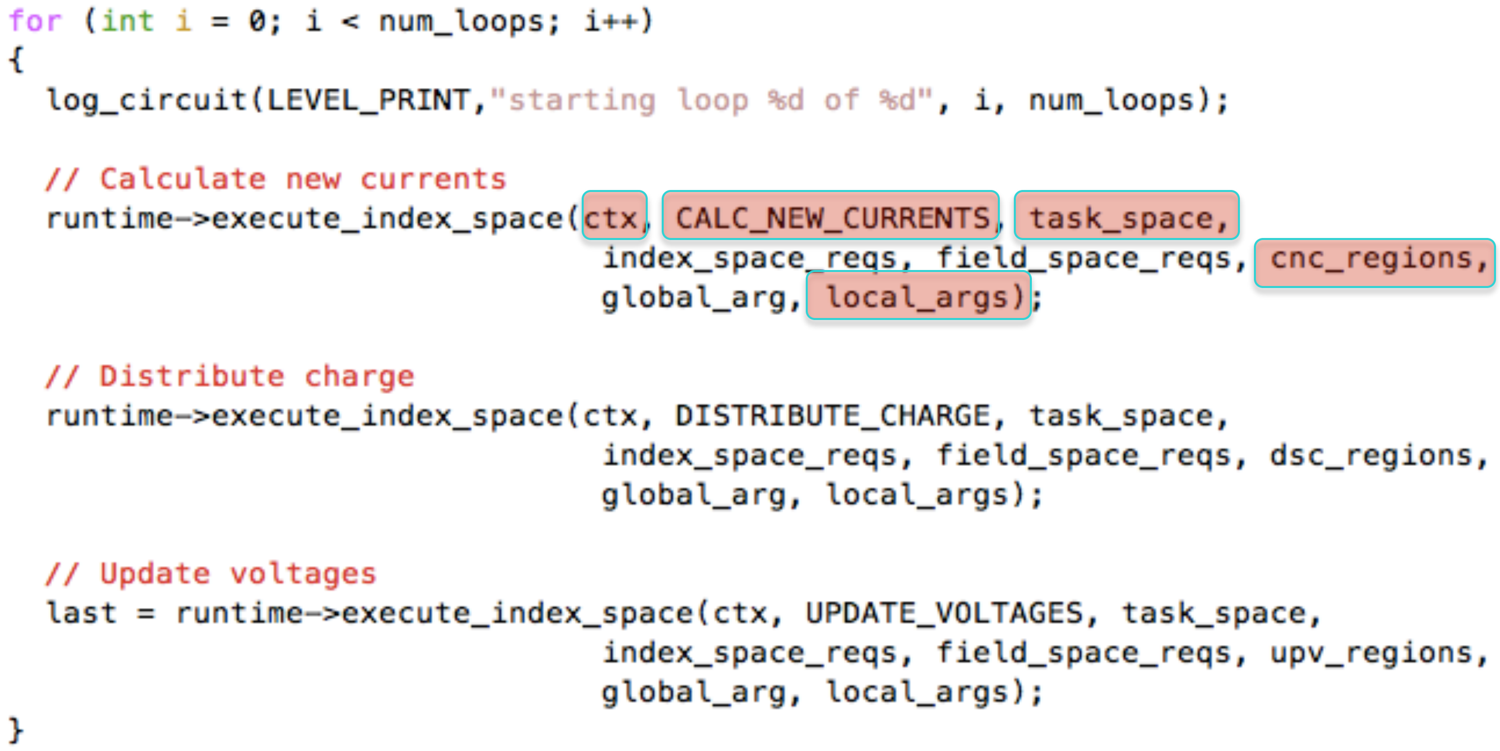
- A *runtime system*
  - All decisions are made dynamically
  - Again, see discussion of Legion compiler …



```
for (int i = 0; i < num_loops; i++)
{
  log_circuit(LEVEL_PRINT,"starting loop %d of %d", i, num_loops);

  // Calculate new currents
  runtime->execute_index_space(ctx, CALC_NEW_CURRENTS, task_space,
                               index_space_reqs, field_space_reqs, cnc_regions,
                               global_arg, local_args);

  // Distribute charge
  runtime->execute_index_space(ctx, DISTRIBUTE_CHARGE, task_space,
                               index_space_reqs, field_space_reqs, dsc_regions,
                               global_arg, local_args);

  // Update voltages
  last = runtime->execute_index_space(ctx, UPDATE_VOLTAGES, task_space,
                               index_space_reqs, field_space_reqs, upv_regions,
                               global_arg, local_args);
}
```

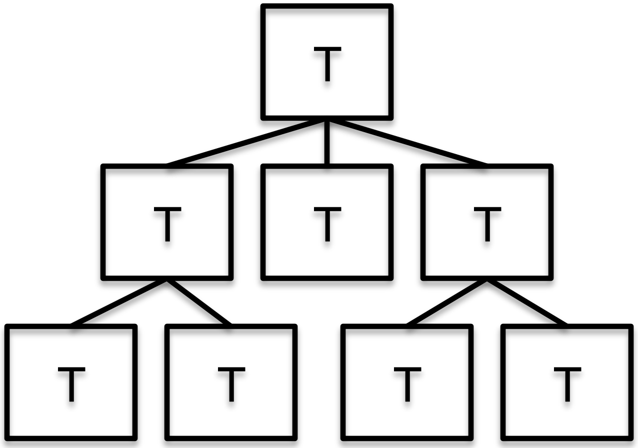
Tasks

- A task is
  - The unit of parallel computation in Legion
  - Takes *regions* (typed collections) as arguments
  - Can launch subtasks



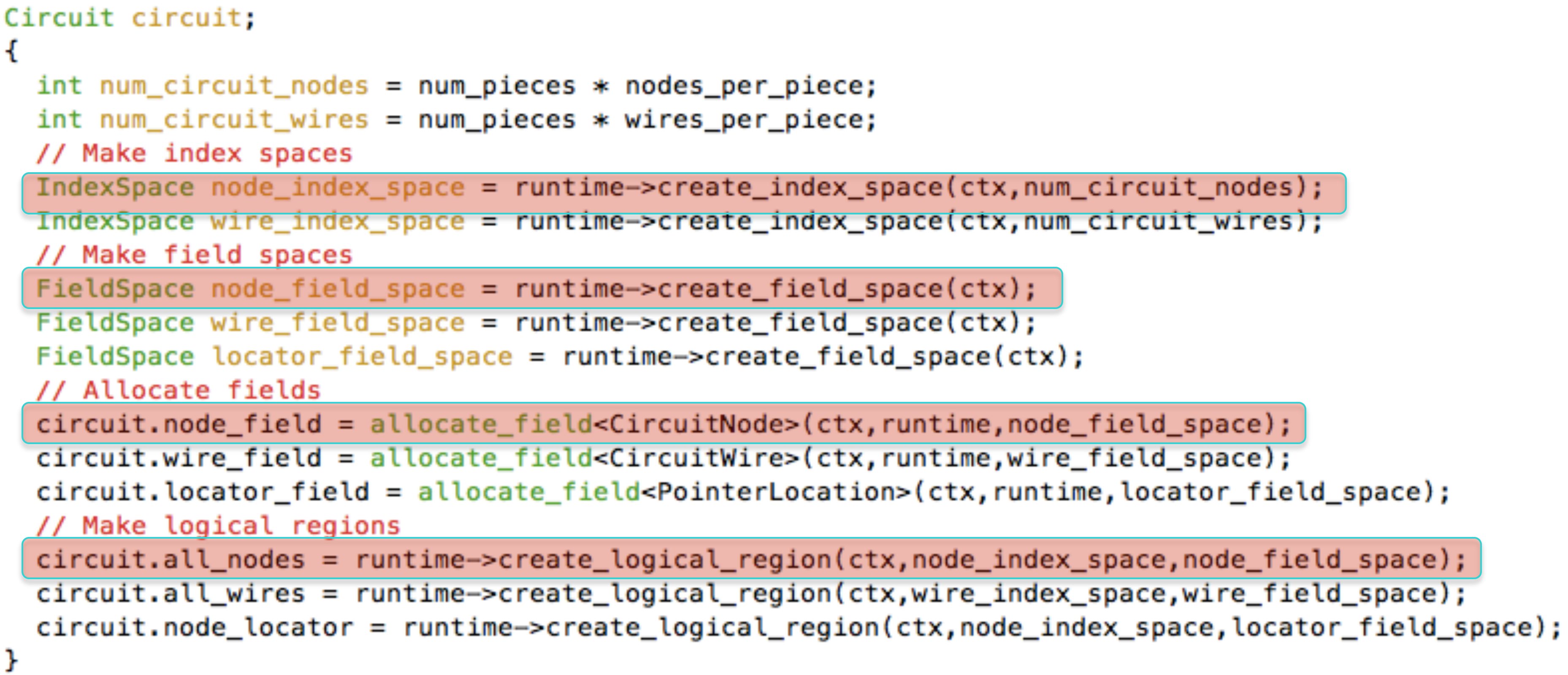
```
for (int i = 0; i < num_loops; i++)
{
  log_circuit(LEVEL_PRINT,"starting loop %d of %d", i, num_loops);

  // Calculate new currents
  runtime->execute_index_space(ctx, CALC_NEW_CURRENTS, task_space,
                               index_space_reqs, field_space_reqs, cnc_regions,
                               global_arg, local_args);

  // Distribute charge
  runtime->execute_index_space(ctx, DISTRIBUTE_CHARGE, task_space,
                               index_space_reqs, field_space_reqs, dsc_regions,
                               global_arg, local_args);

  // Update voltages
  last = runtime->execute_index_space(ctx, UPDATE_VOLTAGES, task_space,
                               index_space_reqs, field_space_reqs, upv_regions,
                               global_arg, local_args);
}
```

Task Tree

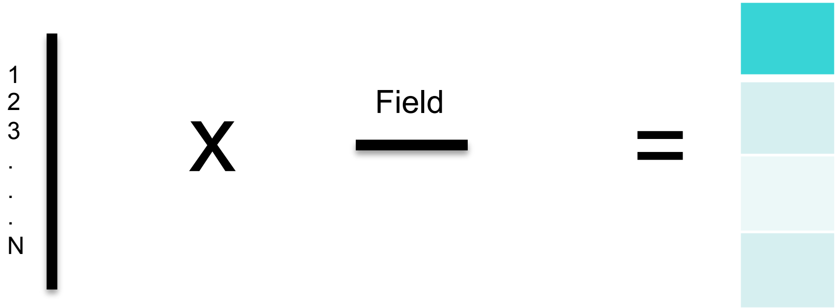


- Legion programs can launch arbitrary trees of tasks
- By default, execute in the order launched
- Runtime system automatically identifies parallel tasks

Regions: Index & Field Spaces



```
Circuit circuit;
{
  int num_circuit_nodes = num_pieces * nodes_per_piece;
  int num_circuit_wires = num_pieces * wires_per_piece;
  // Make index spaces
  IndexSpace node_index_space = runtime->create_index_space(ctx,num_circuit_nodes);
  IndexSpace wire_index_space = runtime->create_index_space(ctx,num_circuit_wires);
  // Make field spaces
  FieldSpace node_field_space = runtime->create_field_space(ctx);
  FieldSpace wire_field_space = runtime->create_field_space(ctx);
  FieldSpace locator_field_space = runtime->create_field_space(ctx);
  // Allocate fields
  circuit.node_field = allocate_field<CircuitNode>(ctx,runtime,node_field_space);
  circuit.wire_field = allocate_field<CircuitWire>(ctx,runtime,wire_field_space);
  circuit.locator_field = allocate_field<PointerLocation>(ctx,runtime,locator_field_space);
  // Make logical regions
  circuit.all_nodes = runtime->create_logical_region(ctx,node_index_space,node_field_space);
  circuit.all_wires = runtime->create_logical_region(ctx,wire_index_space,wire_field_space);
  circuit.node_locator = runtime->create_logical_region(ctx,node_index_space,locator_field_space);
}
```

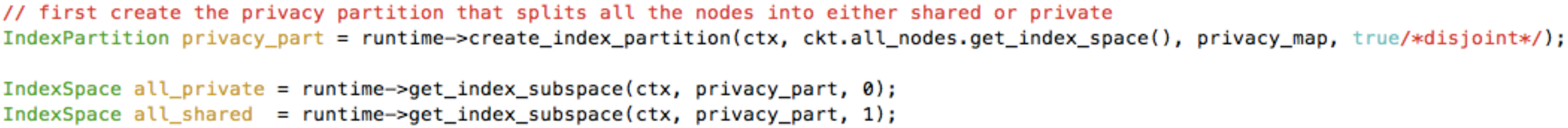


Regions

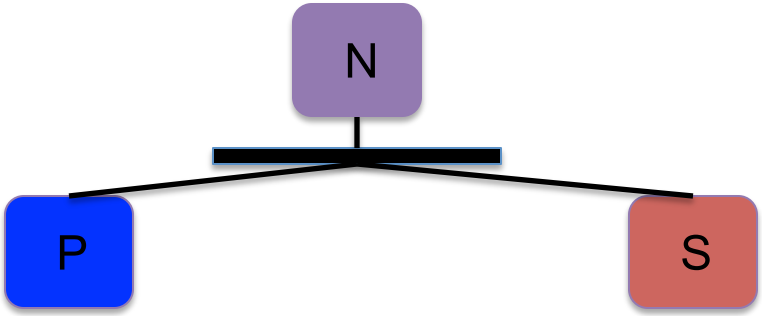
- **Two Dimensions**
- **Unbounded set of rows**
- **Bounded set of columns**
  - **Fields**
- **Tasks declare**
  - **Which fields they use**
  - **And how they use them**
- **Regions can be *partitioned***

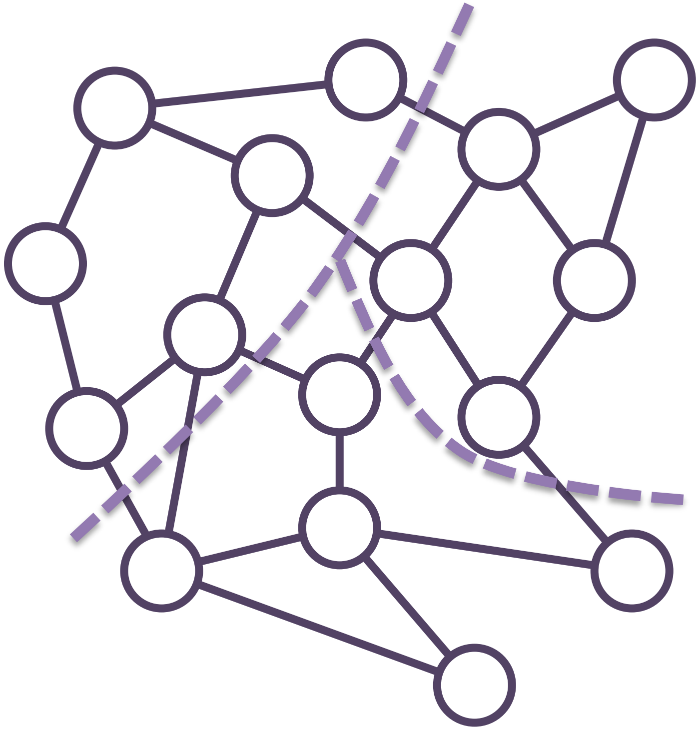
	Voltage	Capac.	Induct.	Charge
Node				
Node				
Node				
Node				
Node				
Node				
Node				
Node				
Node				
Node				

Partitioning

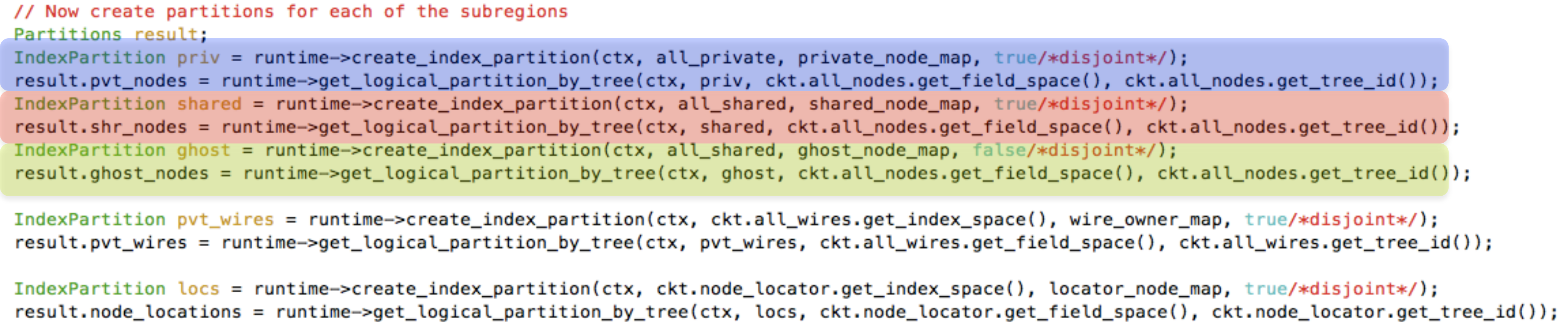


```
// first create the privacy partition that splits all the nodes into either shared or private
IndexPartition privacy_part = runtime->create_index_partition(ctx, ckt.all_nodes.get_index_space(), privacy_map, true/*disjoint*/);

IndexSpace all_private = runtime->get_index_subspace(ctx, privacy_part, 0);
IndexSpace all_shared  = runtime->get_index_subspace(ctx, privacy_part, 1);
```

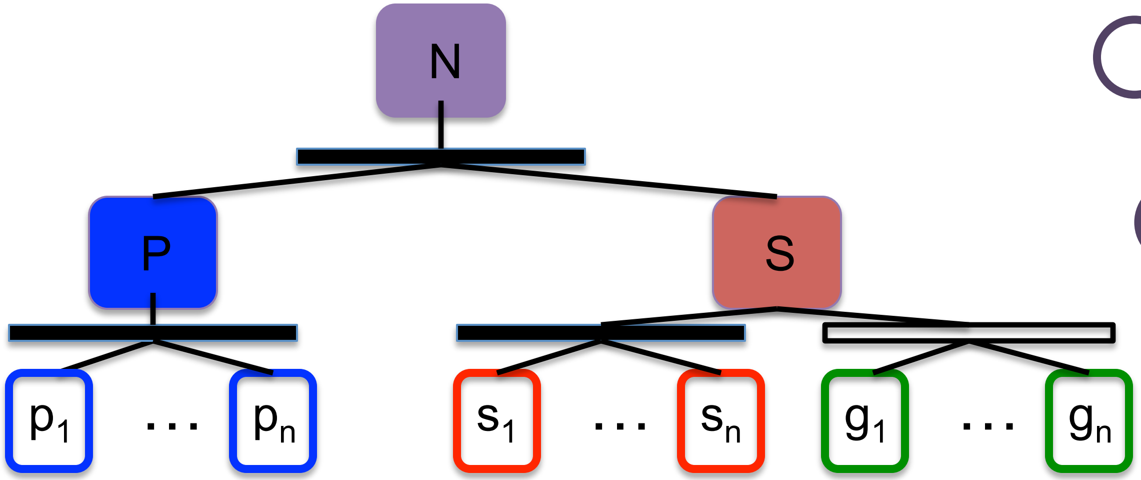


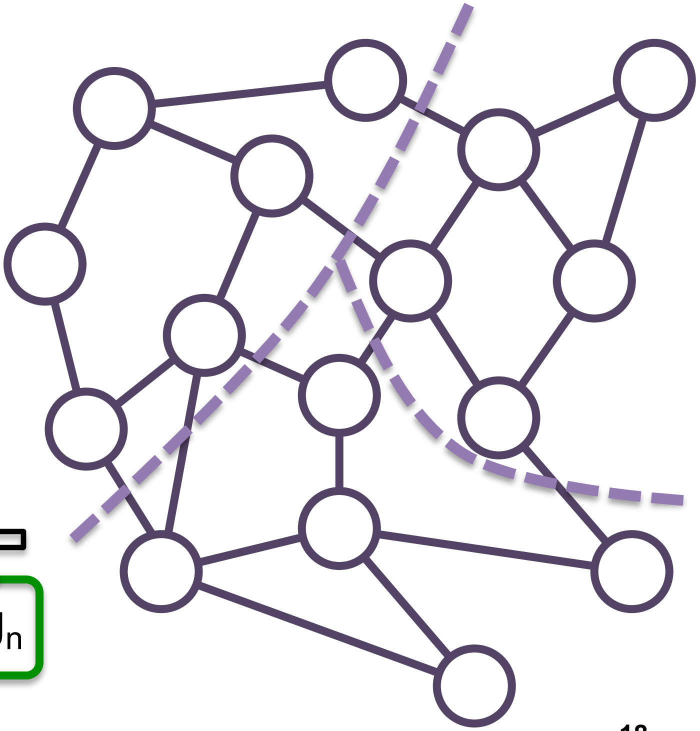


Partitioning



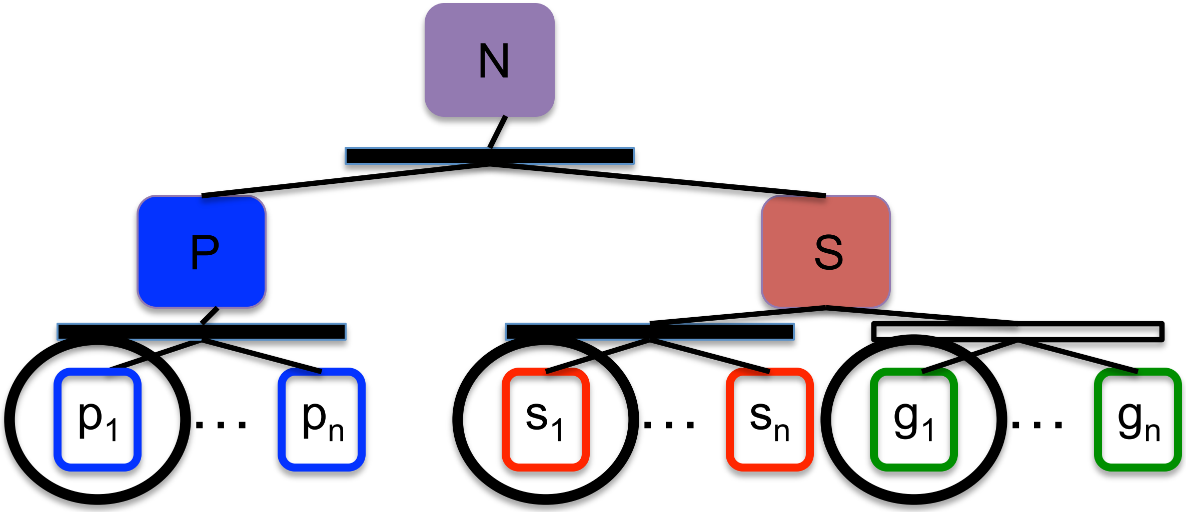
```
// Now create partitions for each of the subregions
Partitions result;
IndexPartition priv = runtime->create_index_partition(ctx, all_private, private_node_map, true/*disjoint*/);
result.pvt_nodes = runtime->get_logical_partition_by_tree(ctx, priv, ckt.all_nodes.get_field_space(), ckt.all_nodes.get_tree_id());
IndexPartition shared = runtime->create_index_partition(ctx, all_shared, shared_node_map, true/*disjoint*/);
result.shr_nodes = runtime->get_logical_partition_by_tree(ctx, shared, ckt.all_nodes.get_field_space(), ckt.all_nodes.get_tree_id());
IndexPartition ghost = runtime->create_index_partition(ctx, all_shared, ghost_node_map, false/*disjoint*/);
result.ghost_nodes = runtime->get_logical_partition_by_tree(ctx, ghost, ckt.all_nodes.get_field_space(), ckt.all_nodes.get_tree_id());

IndexPartition pvt_wires = runtime->create_index_partition(ctx, ckt.all_wires.get_index_space(), wire_owner_map, true/*disjoint*/);
result.pvt_wires = runtime->get_logical_partition_by_tree(ctx, pvt_wires, ckt.all_wires.get_field_space(), ckt.all_wires.get_tree_id());

IndexPartition locs = runtime->create_index_partition(ctx, ckt.node_locator.get_index_space(), locator_node_map, true/*disjoint*/);
result.node_locations = runtime->get_logical_partition_by_tree(ctx, locs, ckt.node_locator.get_field_space(), ckt.node_locator.get_tree_id());
```



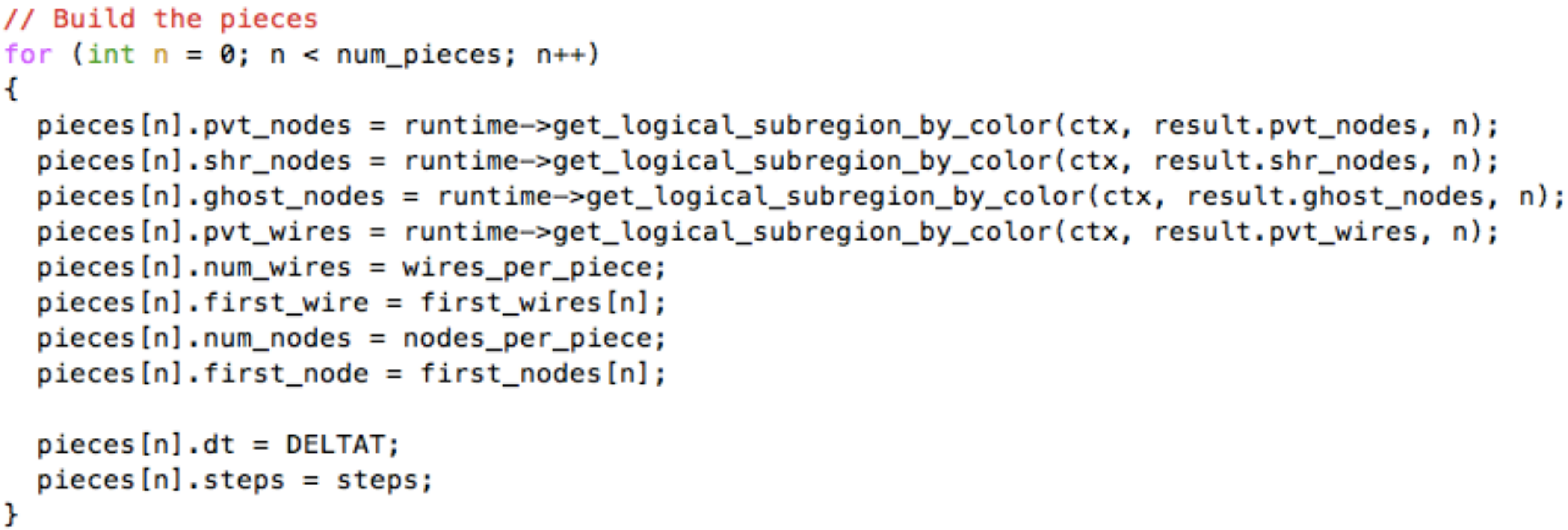


Organize Into Pieces

```
// Build the pieces
for (int n = 0; n < num_pieces; n++)
{
  pieces[n].pvt_nodes = runtime->get_logical_subregion_by_color(ctx, result.pvt_nodes, n);
  pieces[n].shr_nodes = runtime->get_logical_subregion_by_color(ctx, result.shr_nodes, n);
  pieces[n].ghost_nodes = runtime->get_logical_subregion_by_color(ctx, result.ghost_nodes, n);
  pieces[n].pvt_wires = runtime->get_logical_subregion_by_color(ctx, result.pvt_wires, n);
  pieces[n].num_wires = wires_per_piece;
  pieces[n].first_wire = first_wires[n];
  pieces[n].num_nodes = nodes_per_piece;
  pieces[n].first_node = first_nodes[n];

  pieces[n].dt = DELTAT;
  pieces[n].steps = steps;
}
```



Embedded in C++



```
// Build the pieces
for (int n = 0; n < num_pieces; n++)
{
  pieces[n].pvt_nodes = runtime->get_logical_subregion_by_color(ctx, result.pvt_nodes, n);
  pieces[n].shr_nodes = runtime->get_logical_subregion_by_color(ctx, result.shr_nodes, n);
  pieces[n].ghost_nodes = runtime->get_logical_subregion_by_color(ctx, result.ghost_nodes, n);
  pieces[n].pvt_wires = runtime->get_logical_subregion_by_color(ctx, result.pvt_wires, n);
  pieces[n].num_wires = wires_per_piece;
  pieces[n].first_wire = first_wires[n];
  pieces[n].num_nodes = nodes_per_piece;
  pieces[n].first_node = first_nodes[n];

  pieces[n].dt = DELTAT;
  pieces[n].steps = steps;
}
```

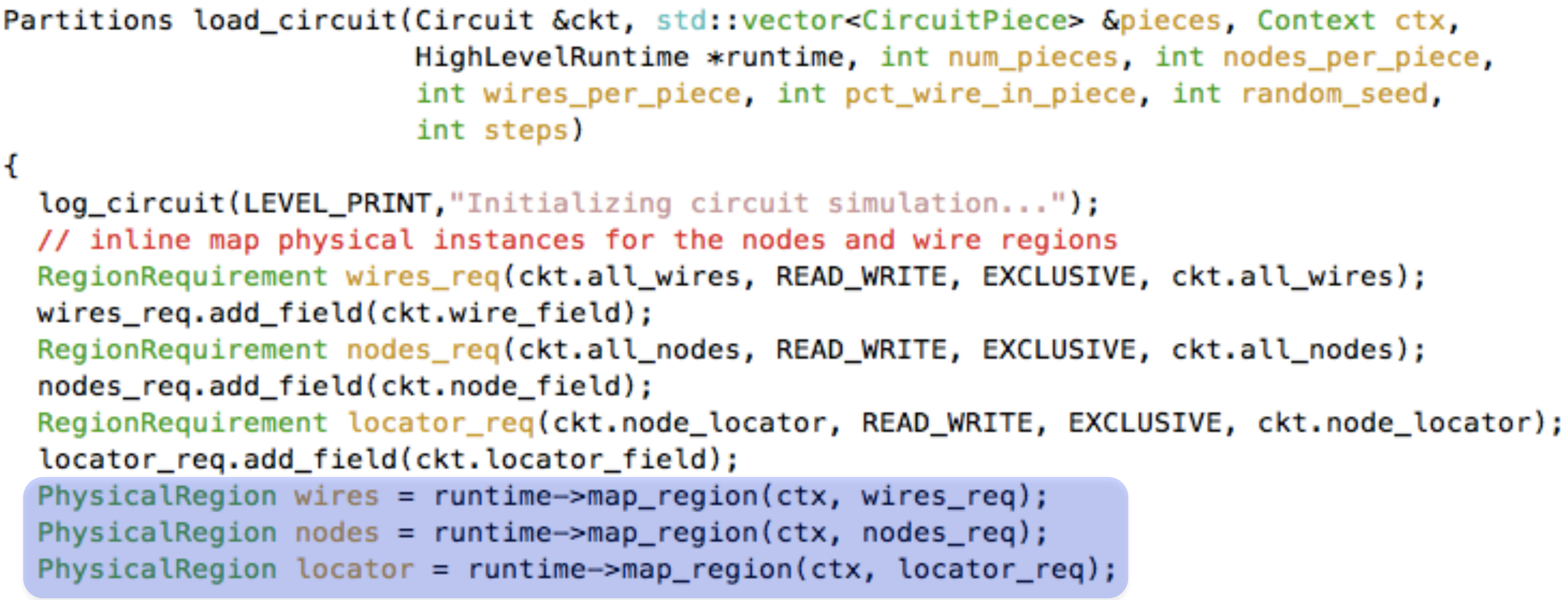
Can write any C++ code within a task

  - Local state, pointers, etc.
  - Must follow discipline when using Legion API

- Regions are first class
  - Can be passed as arguments, stored in data structures

Populating Regions

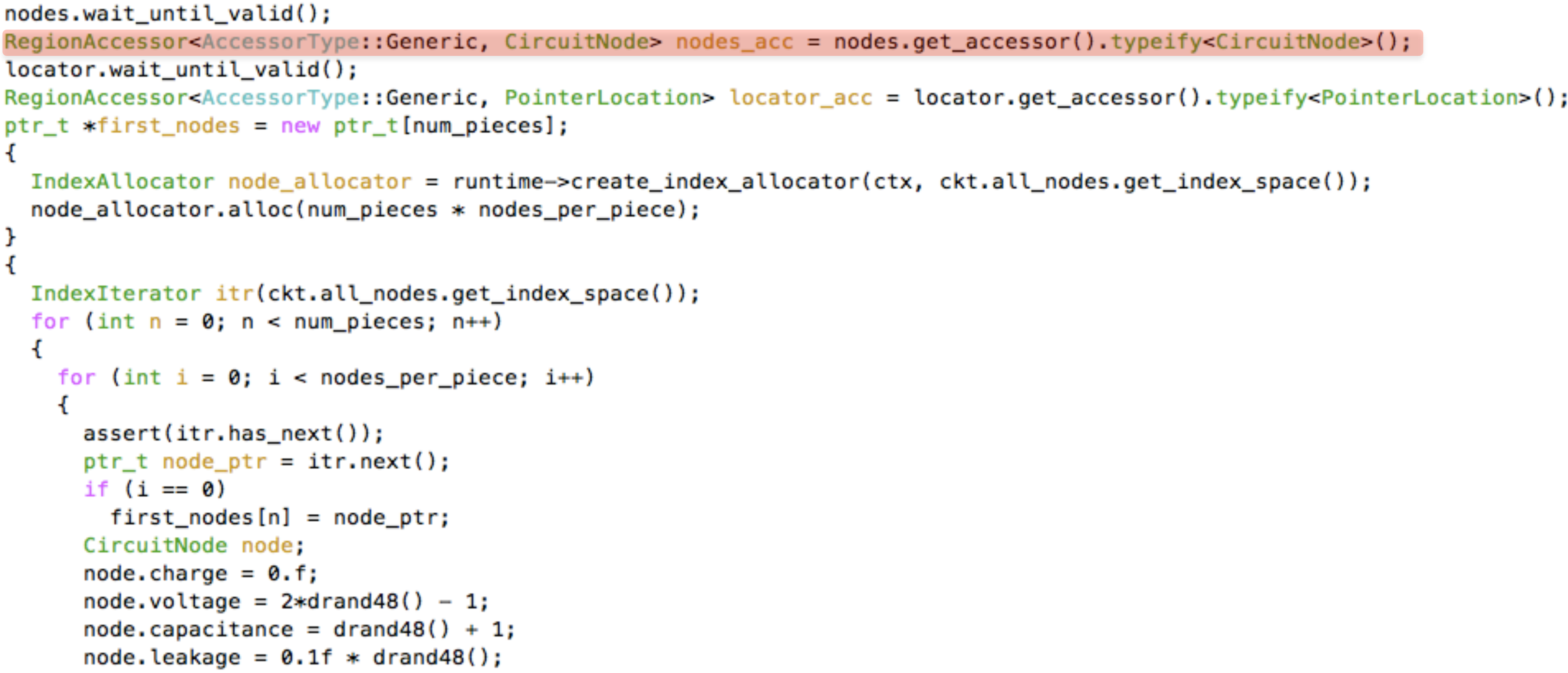
- Can’t read/update a region without an *instance*
  - Instances hold a valid current copy of the data



```
Partitions load_circuit(Circuit &ckt, std::vector<CircuitPiece> &pieces, Context ctx,
                        HighLevelRuntime *runtime, int num_pieces, int nodes_per_piece,
                        int wires_per_piece, int pct_wire_in_piece, int random_seed,
                        int steps)
{
  log_circuit(LEVEL_PRINT,"Initializing circuit simulation...");
  // inline map physical instances for the nodes and wire regions
  RegionRequirement wires_req(ckt.all_wires, READ_WRITE, EXCLUSIVE, ckt.all_wires);
  wires_req.add_field(ckt.wire_field);
  RegionRequirement nodes_req(ckt.all_nodes, READ_WRITE, EXCLUSIVE, ckt.all_nodes);
  nodes_req.add_field(ckt.node_field);
  RegionRequirement locator_req(ckt.node_locator, READ_WRITE, EXCLUSIVE, ckt.node_locator);
  locator_req.add_field(ckt.locator_field);
  PhysicalRegion wires = runtime->map_region(ctx, wires_req);
  PhysicalRegion nodes = runtime->map_region(ctx, nodes_req);
  PhysicalRegion locator = runtime->map_region(ctx, locator_req);
```

Populating Regions

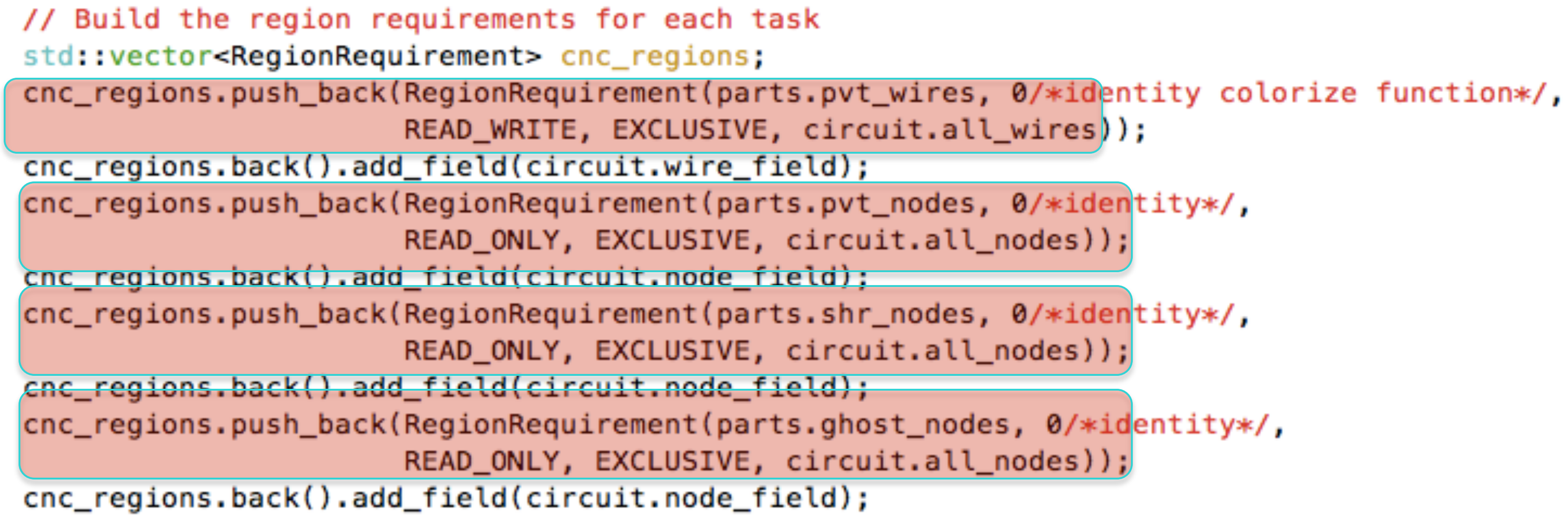
- To read/update a region, need an *accessor*
  - A handle to reference, or iterate through, elements



```
nodes.wait_until_valid();
RegionAccessor<AccessorType::Generic, CircuitNode> nodes_acc = nodes.get_accessor().typeify<CircuitNode>();
locator.wait_until_valid();
RegionAccessor<AccessorType::Generic, PointerLocation> locator_acc = locator.get_accessor().typeify<PointerLocation>();
ptr_t *first_nodes = new ptr_t[num_pieces];
{
  IndexAllocator node_allocator = runtime->create_index_allocator(ctx, ckt.all_nodes.get_index_space());
  node_allocator.alloc(num_pieces * nodes_per_piece);
}
{
  IndexIterator itr(ckt.all_nodes.get_index_space());
  for (int n = 0; n < num_pieces; n++)
  {
    for (int i = 0; i < nodes_per_piece; i++)
    {
      assert(itr.has_next());
      ptr_t node_ptr = itr.next();
      if (i == 0)
        first_nodes[n] = node_ptr;
      CircuitNode node;
      node.charge = 0.f;
      node.voltage = 2*drand48() - 1;
      node.capacitance = drand48() + 1;
      node.leakage = 0.1f * drand48();
```

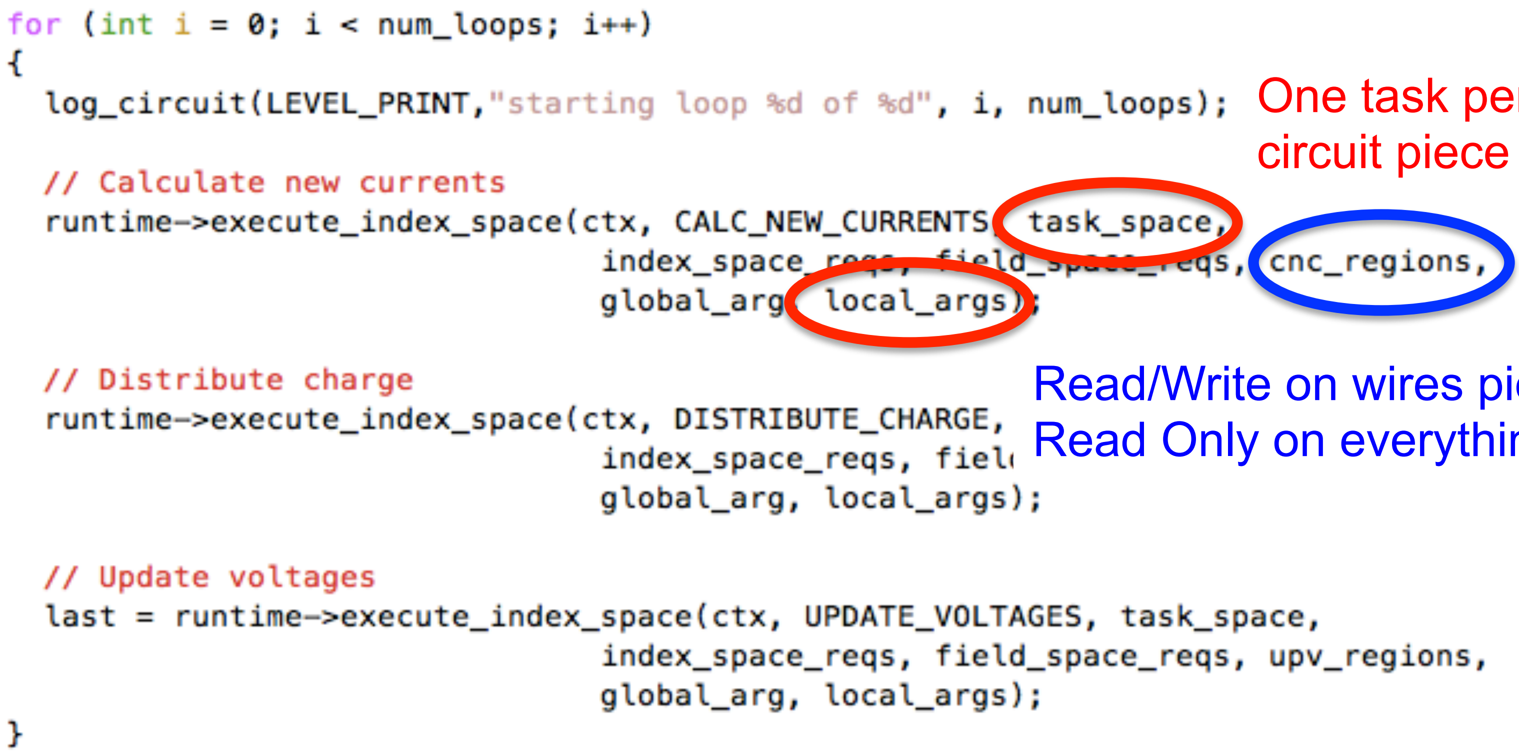
Regions: Privileges & Coherence

```
void calc_new_currents(CircuitPiece piece):
        RWE(piece.rw_pvt), ROE(piece.rn_pvt, piece.rn_shr, piece.rn_ghost) {
  foreach(w : piece.rw_pvt)
    w→current = (w→in_node→voltage − w→out_node→voltage) / w→resistance;
}
```



```
// Build the region requirements for each task
std::vector<RegionRequirement> cnc_regions;
cnc_regions.push_back(RegionRequirement(parts.pvt_wires, 0/*identity colorize function*/,
                                         READ_WRITE, EXCLUSIVE, circuit.all_wires));
cnc_regions.back().add_field(circuit.wire_field);
cnc_regions.push_back(RegionRequirement(parts.pvt_nodes, 0/*identity*/,
                                         READ_ONLY, EXCLUSIVE, circuit.all_nodes));
cnc_regions.back().add_field(circuit.node_field);
cnc_regions.push_back(RegionRequirement(parts.shr_nodes, 0/*identity*/,
                                         READ_ONLY, EXCLUSIVE, circuit.all_nodes));
cnc_regions.back().add_field(circuit.node_field);
cnc_regions.push_back(RegionRequirement(parts.ghost_nodes, 0/*identity*/,
                                         READ_ONLY, EXCLUSIVE, circuit.all_nodes));
cnc_regions.back().add_field(circuit.node_field);
```

Back to the Simulation



```
for (int i = 0; i < num_loops; i++)
{
  log_circuit(LEVEL_PRINT,"starting loop %d of %d", i, num_loops);

  // Calculate new currents
  runtime->execute_index_space(ctx, CALC_NEW_CURRENTS, task_space,
                               index_space_reqs, field_space_reqs, cnc_regions,
                               global_arg, local_args);

  // Distribute charge
  runtime->execute_index_space(ctx, DISTRIBUTE_CHARGE,
                               index_space_reqs, fiel
                               global_arg, local_args);

  // Update voltages
  last = runtime->execute_index_space(ctx, UPDATE_VOLTAGES, task_space,
                               index_space_reqs, field_space_reqs, upv_regions,
                               global_arg, local_args);
}
```

One task per circuit piece

Read/Write on wires pieces
Read Only on everything else

The Crux

- Crucial design decisions in a Legion program are:

- What are the regions?
  - How are the regions partitioned into subregions?

- What are the tasks?
  - How are the tasks decomposed into subtasks?

- Often tension between the two
  - These decisions drive the program's design

Summary

- The programmer
  - Describes the structure of the program's data
    - Regions
  - The tasks that operate on that data

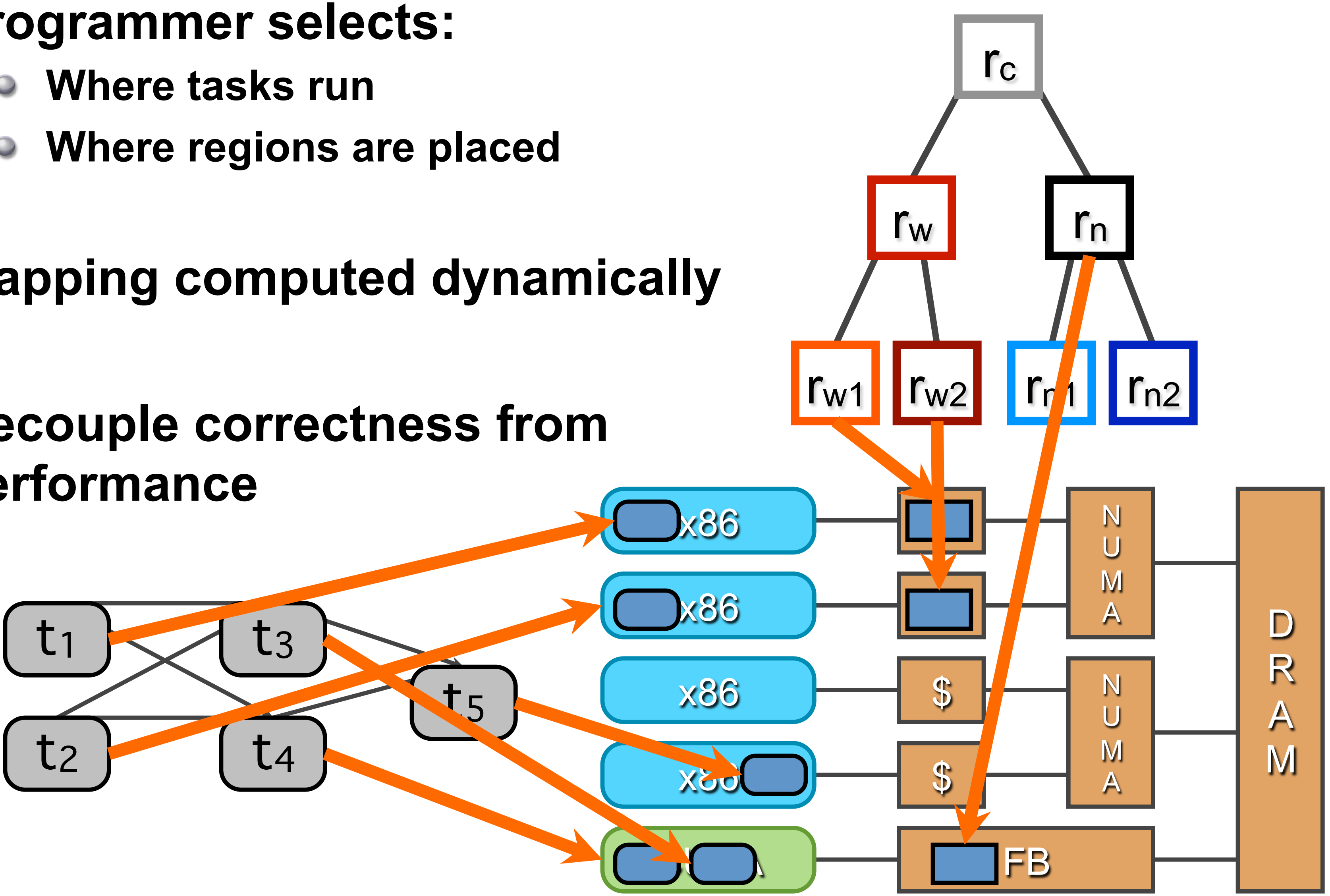
- The Legion implementation
  - Guarantees tasks appear to execute in sequential order
  - Ensures tasks have valid versions of their regions

Mapping Interface

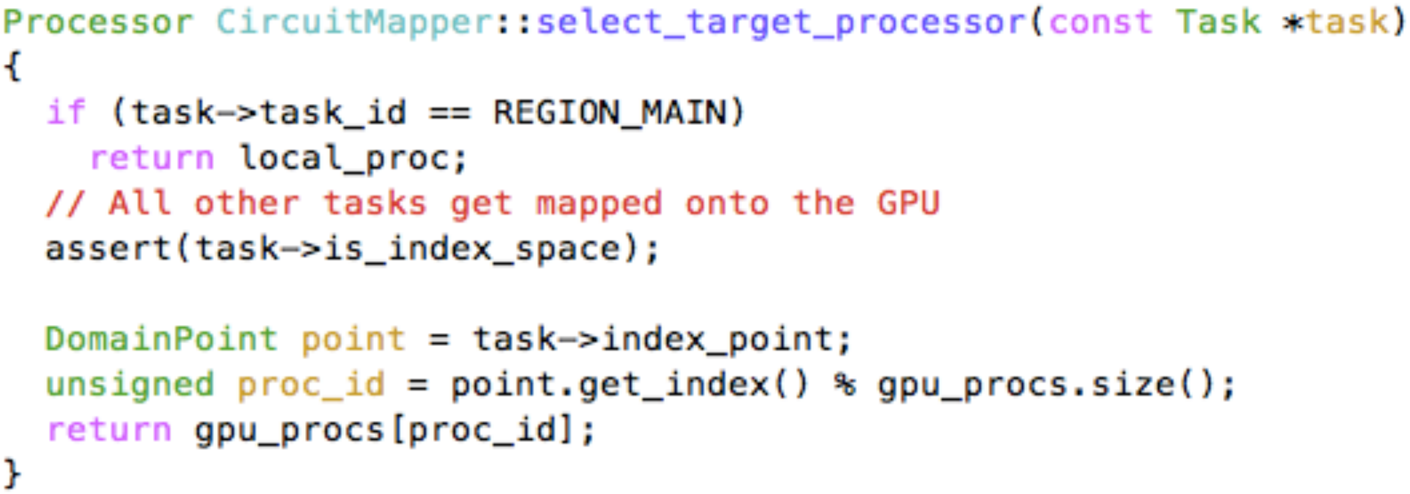
- Programmer selects:
  - Where tasks run
  - Where regions are placed

- Mapping computed dynamically

- Decouple correctness from performance



Mapping



```
Processor CircuitMapper::select_target_processor(const Task *task)
{
  if (task->task_id == REGION_MAIN)
    return local_proc;
  // All other tasks get mapped onto the GPU
  assert(task->is_index_space);

  DomainPoint point = task->index_point;
  unsigned proc_id = point.get_index() % gpu_procs.size();
  return gpu_procs[proc_id];
}
```

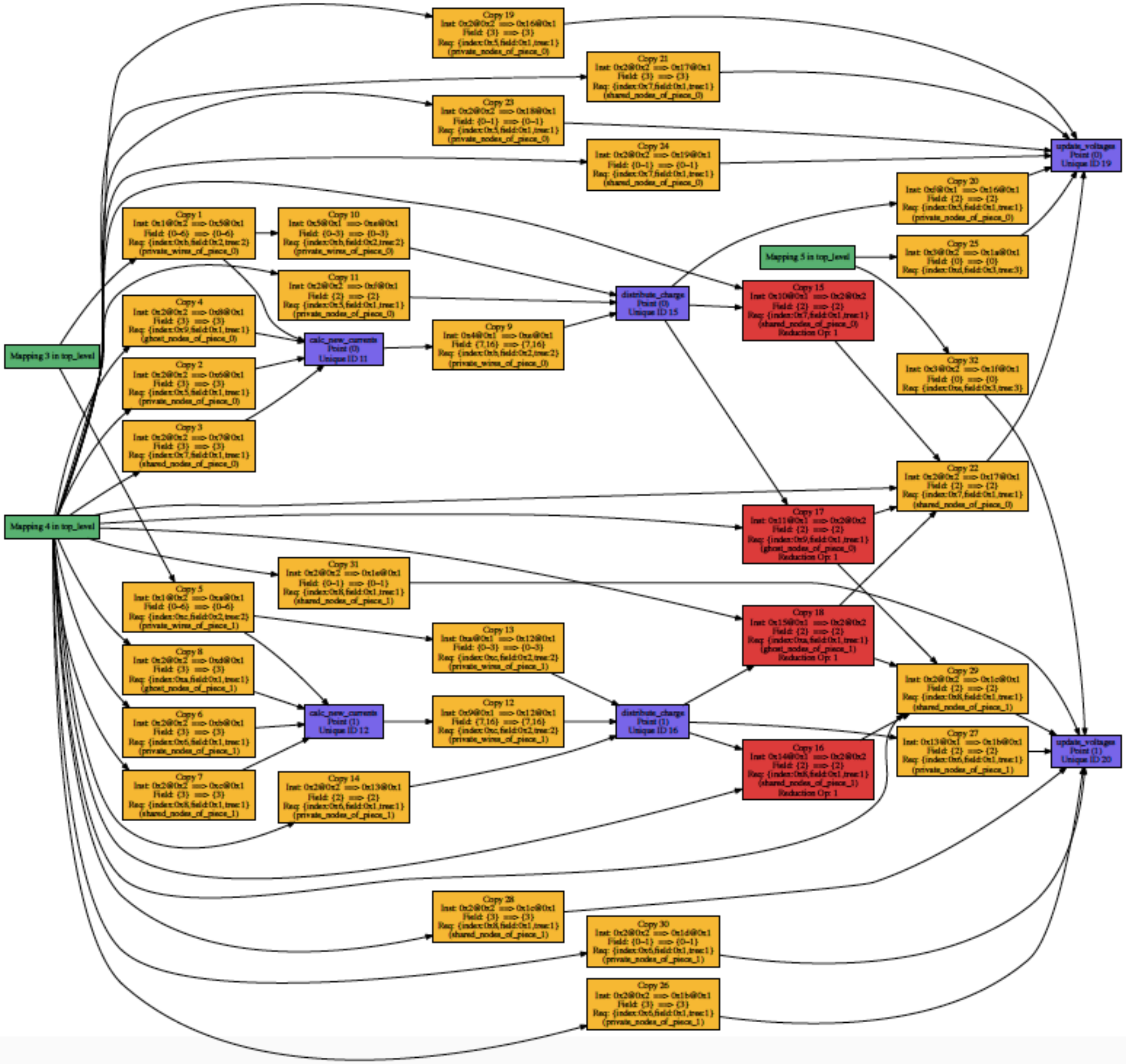
- Mapper interface = callback interface
  - Legion runtime calls user-supplied methods
  - Can do arbitrary computation to make decisions
    - But often very simple

The Crux, Revisited

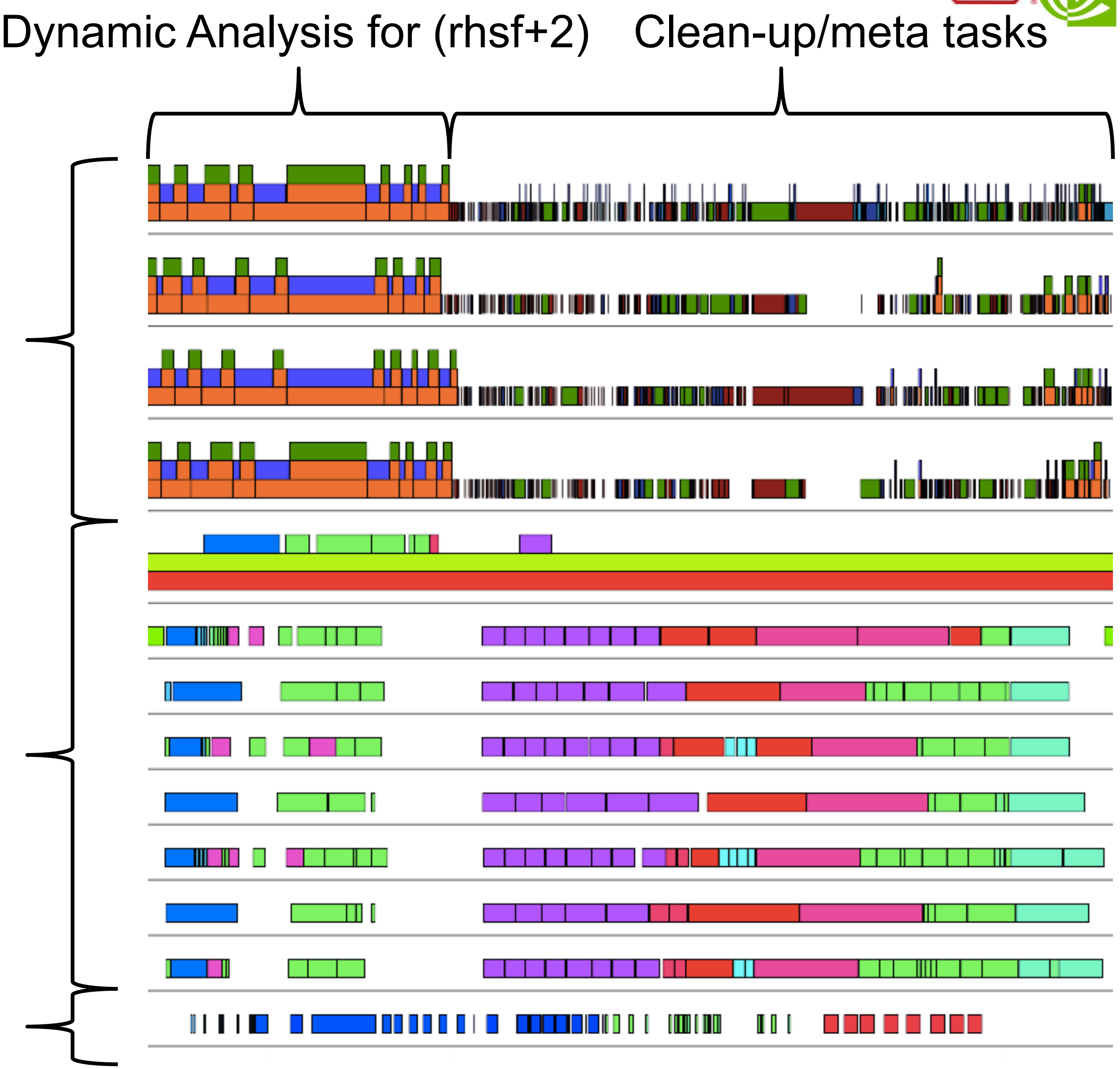
- Crucial design decisions in a Legion program are
  - What are the regions?
  - What are the tasks?

- In particular, mapping decisions depend on design of the regions and tasks
  - Not the other way around

Debugging


Copy 19
Inst: 0x2@0x2 ==> 0x16@0x1
Field: {3} ==> {3}
Req: {index:0x5,field:0x1,tree:1}
(private_nodes_of_piece_0)
Copy 21
Inst: 0x2@0x2 ==> 0x17@0x1
Field: {3} ==> {3}
Req: {index:0x7,field:0x1,tree:1}
(shared_nodes_of_piece_0)
Copy 23
Inst: 0x2@0x2 ==> 0x18@0x1
Field: {0–1} ==> {0–1}
Req: {index:0x5,field:0x1,tree:1}
(private_nodes_of_piece_0)
Copy 24
Inst: 0x2@0x2 ==> 0x19@0x1
Field: {0–1} ==> {0–1}
Req: {index:0x7,field:0x1,tree:1}
(shared_nodes_of_piece_0)
update_voltages
Point (0)
Unique ID 19
Copy 20
Inst: 0xf@0x1 ==> 0x16@0x1
Field: {2} ==> {2}
Req: {index:0x5,field:0x1,tree:1}
(private_nodes_of_piece_0)
Copy 1
Inst: 0x1@0x2 ==> 0x5@0x1
Field: {0–6} ==> {0–6}
Req: {index:0xb,field:0x2,tree:2}
(private_wires_of_piece_0)
Copy 10
Inst: 0x5@0x1 ==> 0xa@0x1
Field: {0–3} ==> {0–3}
Req: {index:0xb,field:0x2,tree:2}
(private_wires_of_piece_0)
Mapping 5 in top_level
Copy 25
Inst: 0x3@0x2 ==> 0x1a@0x1
Field: {0} ==> {0}
Req: {index:0xd,field:0x3,tree:3}
Copy 11
Inst: 0x2@0x2 ==> 0xf@0x1
Field: {2} ==> {2}
Req: {index:0x5,field:0x1,tree:1}
(private_nodes_of_piece_0)
distribute_charge
Point (0)
Unique ID 15
Copy 15
Inst: 0x10@0x1 ==> 0x2@0x2
Field: {2} ==> {2}
Req: {index:0x7,field:0x1,tree:1}
(shared_nodes_of_piece_0)
Reduction Op: 1
Copy 4
Inst: 0x2@0x2 ==> 0x8@0x1
Field: {3} ==> {3}
Req: {index:0x9,field:0x1,tree:1}
(ghost_nodes_of_piece_0)
calc_new_currents
Point (0)
Unique ID 11
Copy 9
Inst: 0x4@0x1 ==> 0xa@0x1
Field: {7,16} ==> {7,16}
Req: {index:0xb,field:0x2,tree:2}
(private_wires_of_piece_0)
Mapping 3 in top_level
Copy 2
Inst: 0x2@0x2 ==> 0x6@0x1
Field: {3} ==> {3}
Req: {index:0x5,field:0x1,tree:1}
(private_nodes_of_piece_0)
Copy 32
Inst: 0x3@0x2 ==> 0x1f@0x1
Field: {0} ==> {0}
Req: {index:0xe,field:0x3,tree:3}
Copy 3
Inst: 0x2@0x2 ==> 0x7@0x1
Field: {3} ==> {3}
Req: {index:0x7,field:0x1,tree:1}
(shared_nodes_of_piece_0)
Copy 22
Inst: 0x2@0x2 ==> 0x17@0x1
Field: {2} ==> {2}
Req: {index:0x7,field:0x1,tree:1}
(shared_nodes_of_piece_0)
Copy 17
Inst: 0x11@0x1 ==> 0x2@0x2
Field: {2} ==> {2}
Req: {index:0x9,field:0x1,tree:1}
(ghost_nodes_of_piece_0)
Reduction Op: 1
Mapping 4 in top_level
Copy 31
Inst: 0x2@0x2 ==> 0x1e@0x1
Field: {0–1} ==> {0–1}
Req: {index:0x8,field:0x1,tree:1}
(shared_nodes_of_piece_1)
Copy 5
Inst: 0x1@0x2 ==> 0xa@0x1
Field: {0–6} ==> {0–6}
Req: {index:0xc,field:0x2,tree:2}
(private_wires_of_piece_1)
Copy 18
Inst: 0x15@0x1 ==> 0x2@0x2
Field: {2} ==> {2}
Req: {index:0xa,field:0x1,tree:1}
(ghost_nodes_of_piece_1)
Reduction Op: 1
Copy 13
Inst: 0xa@0x1 ==> 0x12@0x1
Field: {0–3} ==> {0–3}
Req: {index:0xc,field:0x2,tree:2}
(private_wires_of_piece_1)
Copy 8
Inst: 0x2@0x2 ==> 0xd@0x1
Field: {3} ==> {3}
Req: {index:0xa,field:0x1,tree:1}
(ghost_nodes_of_piece_1)
Copy 29
Inst: 0x2@0x2 ==> 0x1c@0x1
Field: {2} ==> {2}
Req: {index:0x8,field:0x1,tree:1}
(shared_nodes_of_piece_1)
calc_new_currents
Point (1)
Unique ID 12
Copy 12
Inst: 0x9@0x1 ==> 0x12@0x1
Field: {7,16} ==> {7,16}
Req: {index:0xc,field:0x2,tree:2}
(private_wires_of_piece_1)
distribute_charge
Point (1)
Unique ID 16
Copy 6
Inst: 0x2@0x2 ==> 0xb@0x1
Field: {3} ==> {3}
Req: {index:0x6,field:0x1,tree:1}
(private_nodes_of_piece_1)
Copy 27
Inst: 0x13@0x1 ==> 0x1b@0x1
Field: {2} ==> {2}
Req: {index:0x6,field:0x1,tree:1}
(private_nodes_of_piece_1)
update_voltages
Point (1)
Unique ID 20
Copy 16
Inst: 0x14@0x1 ==> 0x2@0x2
Field: {2} ==> {2}
Req: {index:0x8,field:0x1,tree:1}
(shared_nodes_of_piece_1)
Reduction Op: 1
Copy 7
Inst: 0x2@0x2 ==> 0xc@0x1
Field: {3} ==> {3}
Req: {index:0x8,field:0x1,tree:1}
(shared_nodes_of_piece_1)
Copy 14
Inst: 0x2@0x2 ==> 0x13@0x1
Field: {2} ==> {2}
Req: {index:0x6,field:0x1,tree:1}
(private_nodes_of_piece_1)
Copy 28
Inst: 0x2@0x2 ==> 0x1c@0x1
Field: {3} ==> {3}
Req: {index:0x8,field:0x1,tree:1}
(shared_nodes_of_piece_1)
Copy 30
Inst: 0x2@0x2 ==> 0x1d@0x1
Field: {0–1} ==> {0–1}
Req: {index:0x6,field:0x1,tree:1}
(private_nodes_of_piece_1)
Copy 26
Inst: 0x2@0x2 ==> 0x1b@0x1
Field: {3} ==> {3}
Req: {index:0x6,field:0x1,tree:1}
(private_nodes_of_piece_1)

LegionProf (Heptane 48^3)



Dynamic Analysis for (rhsf+2)

Clean-up/meta tasks

4 AMD Interlagos Integer cores for Legion Runtime

8 AMD Interlagos FP cores for application

NVIDIA Kepler K20

http://legion.stanford.edu

Questions?