Legion Bootcamp: Data Model

Sean Treichler

Tasks Operate on Data

- A task has a stack and heap – used for private (intra-task) data
- Nearly all data shared between tasks lives in *logical regions*
- Logical regions are explicitly created (and destroyed) by app
- Tasks are annotated with which regions (or parts of regions) they will access

Logical Regions

- A logical region is a collection of elements, each of which has data stored in one or more fields
  - All elements in a region use the same set of fields
- Element named by its index, which is constant for lifetime of element
  - Can be safely stored as a “pointer” in other data structures
- Logical regions are “logical” because:
  - They are not fixed in a particular memory – they can migrate, or be sharded or replicated
  - They don’t have a fixed layout in memory
  - Decisions about placement and layout are made by mapper without changes to app code

Logical Regions are like...

- Arrays of structures:

  ```
  region[index].field = value
  ```

  - Efficient address calculation
  - Compact storage

- Relational database tables:

  ```
  update region set field = value where key = index;
  ```

  - Efficient views
  - Projections
  - Some join-like operations
  - Replication/sharding

Creating a Logical Region

```
LogicalRegion lr = runtime->create_logical_region(ctx, is, fs);
```

- To create a logical region, you must provide:
  - An index space – the "names" of elements in the logical region
  - A field space – the set of fields that will exist for each element
- Runtime returns a logical region handle that may be:
  - Immediately used in the current task
  - Used by any child task launched after the region is created
  - Used by the parent task after the current task has finished
- Can create multiple logical regions with the same index and/or field spaces
  - Each has the same "structure", but distinct data

Index spaces

- Index spaces may be unstructured or structured
- Unstructured index spaces:
  - use opaque ptr_t as index
  - support dynamic alloc/free of elements
  - may be sparse
- Structured index spaces:
  - defined over N-dimensional rectangle (currently N <= 3)
  - use N-dimensional point as index
  - currently always dense (no alloc/free)
- Plan is to unify these over time
  - e.g. support for sparse matrices

Unstructured Index Space Example

```
// for now, have to place an upper bound on number of elements
IndexSpace is = runtime->create_index_space(ctx, max_num_elmts);

{
  IndexAllocator isa = runtime->create_index_space_allocator(ctx, is);

  // can alloc and free individual elements
  ptr_t p = isa.alloc();
  ...
  isa.free(p);

  // can also alloc and free in bulk
  int count = 200;
  ptr_t ps = isa.alloc(count);
  ...
  isa.free(ps, count);
}
```

Structured Index Space Example

```
// create a Rect<N> describing the range we want to cover
Point<2> lo;  lo.x[0] = 10;  lo.x[1] = -100;
Point<2> hi;  hi.x[0] = 15;  hi.x[1] = 100;
Rect<2> range(lo, hi); // bounds are both inclusive

IndexSpace is = runtime->create_index_space(ctx,
                                     Domain::from_rect<2>(range));

...

runtime->destroy_index_space(ctx, is);
```

Field Space Example

```
FieldSpace fs = runtime->create_field_space(ctx);

{
  FieldAllocator fsa = runtime->create_field_allocator(ctx, fs);

  // allocate a field with an app-chosen field ID...
  fsa.allocate_field(sizeof(double), FID_MYFIELD);

  // or let the runtime pick an unused ID for you
  FieldID fid = fsa.allocate_field(sizeof(int));

  ...

  fsa.free_field(FID_MYFIELD);
}
```

Data Operations on Elements

- We distinguish between three types of operations on fields of an element:
- Read:

```
value = region[index].field;
```

- Write:

```
region[index].field = value;
```

- Reduction (per-element):

```
region[index].field += value;
region[index].field *= value;
region[index].field = user_fn(region[index].field,
                              value);
```

Accessing a Logical Region

- Can't access a logical region directly... Why?
  - Logical regions don't have a fixed location in memory

- Instead, we need to:
  1) Map the logical region to a physical instance (PhysicalRegion)
  2) Obtain a RegionAccessor for the desired field
  3) Perform data access operations on the RegionAccessor

- Let's go through each of these steps in more detail

Obtaining a PhysicalRegion

- To get a usable physical instance, we need to:
  1) describe the "requirements" - what we need in the instance and what opearations we want to perform
  2) initiate a mapping of the logical region
  3) decide whether to create a new instance or re-use an existing one
  4) decide which of the region's fields should be stored in the instance
  5) decide in which Memory to place the instance
  6) decide how the instance should be laid out – e.g. AOS vs. SOA
  7) determine what task(s) are producing the data we need
  8) determine what copies must be performed
  9) wait until tasks/copies are complete
- A team effort:

  application   runtime   mapper

Region Requirements

- App creates a `RegionRequirement` object which stores:
  - Logical region (or subregion) containing elements that may be accessed
  - Fields that may be accessed
  - Privileges (read, write, reduce) that may be needed
  - Coherence mode (exclusive or relaxed)
  - Parent logical region (to determine available privileges)
- Index space launches can use a set of logical regions and a projection function to map tasks to regions
- Coming soon: ability to request different privileges for different fields

Privilege Containment Property

- A subtask may only request privileges that its parent holds

- or -

- A parent task must request any privileges that may be needed by a subtask

- Allows sound hierarchical reasoning about application
  - A task's privileges bound the effects of it and all possible descendents

- Enables Legion's scalable distributed scheduling algorithm

Requesting the mapping

- Two ways to request a physical instance for a logical region

- Explicit request within a task ("inline mapping")

```
RegionRequirement req(logical_region, ...);
PhysicalRegion pr = runtime->map_region(ctx, req);
// other stuff if you've got anything independent to do
pr.wait_until_valid();
```

- Automatically performed for all* region requirements for a new task

```
launcher.add_region_requirement(req);
runtime->execute_task(ctx, launcher);
...
void my_task(..., std::vector<PhysicalRegion>& regions, ...)
{}
```

- PhysicalRegion handle is valid until end of task or explicit unmap

Obtaining a RegionAccessor

- A PhysicalRegion can have many potential layouts in memory
  - Possibly even different layouts for fields in same instance

- Amortize decisions about how to efficiently access it in the form of a RegionAccessor
  - One for each field

- GenericAccessor supports the following operations:

```
T value = accessor.read(index);
accessor.write(index, new_value);
accessor.reduce<REDOP>(index, reduce_value);
AT specialized_accessor = accessor.convert<AT>();
T *array_ptr = accessor.raw_rect_ptr(...);
```

Efficient access – specialized accessors

- Use convert method to convert a GenericAccessor into one optimized for particular layouts:
  - AOS/SOA<N> - allows array-indexing-like address calculation if stride between adjacent elements is consistent (stride N can be fixed at compile time or determined dynamically)
  - ReductionFold – similar, but for reduction-only instances
  - ReductionList – handles appending to reduction list instance
  - more as needed...

Efficient access – specialized accessors (2)

- Specialized accessors support (often subset of) same read/write/reduce methods
  - Allows code to be templated on accessor type
  - Methods generally inlined to expose optimization opportunities

- Conversion will fail if PhysicalRegion layout is incompatible with specialized accessor
  - Can check first with can_convert() method
  - Or just make sure mapper demands correct layout
  - Soon: Specify constraints so runtime can select correct variant

Efficient access – raw array pointer

- Gets a raw base pointer and strides – you perform addressing computations
  - Useful when interfacing with C/Fortran, or when manual strength reduction optimizations are needed
  - Pointers valid only until PhysicalRegion is unmapped

```
Rect<DIM> in_rect;         // rect for which pointers are requested
Rect<DIM> out_rect;        // subrect where ptrs are actually valid
ByteOffset strides[DIM];   // address stride in each dimension
T *base_ptr;               // address of top-left element
base_ptr = accessor.raw_rect_ptr<DIM>(in_rect, out_rect, strides);

// 1-D iteration
for(int i = out_rect.lo[0]; i <= out_rect.hi[0]; i++) {
  T val = *base_ptr;
  base_ptr += strides[0];
}
```

Precision: Correctness vs. Performance

- Precision in requirements is rewarded
  - Underspecifying requirements leads to runtime errors or race conditions (if runtime checks are disabled)
  - Overspecifying requirements leads to extra copying, reduced parallelism
- Precise specification of fields and privileges isn't too hard
  - Generally easy to determine from static/human analysis of code
  - Conditionals in code can still cause issues though
- Harder problem for specifying which elements of region will be accessed
  - Need efficient way of describing/naming arbitrary subsets of elements in a region

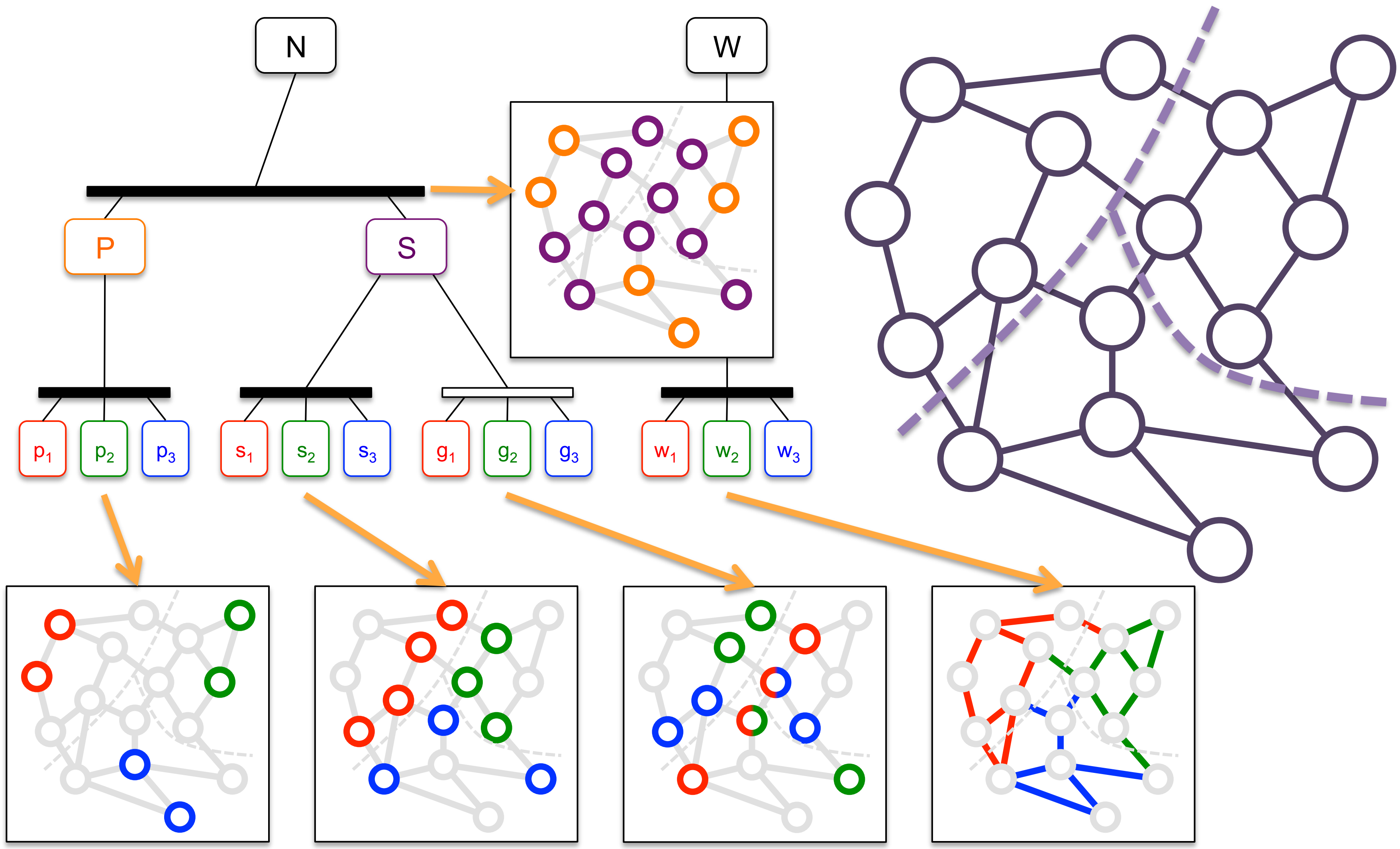
Partitions, Subregions

- A partition gives names to subsets of a logical region
  - Each partition has one or more subregions
  - Subregions are identified by their “color”, a unique index in a “color space” (usually a 1-D index space)
  - The subregions of a partition may be disjoint or aliased
- Subregions may be further partitioned
  - Creates a “region tree”
  - Can help match hierarchical structure of task decomposition
  - Or hierarchical structure of memory
  - Or application communication patterns
  - ...

Multiple Views on the Same Data

- Partitioning names new subsets, but keeps old ones
  - Can use both parent region and subregions in subsequent mappings
  - Runtime understands relationship, manages parallelism/coherence
- Multiple partitions may be defined of the same region
  - Provides multiple (possibly overlapping) views of the same data
  - Each partition also assigned a “color” (an app-specified ID)
- Colors can be used to walk down from root to any partition/ subregion
  - Allows other tasks to find subregions without sending all the handles around

Partitioning Example



A few more details

- Partitioning actually an operation on an IndexSpace
  - Creates an "index tree" of IndexPartitions and IndexSpaces
  - Implicitly defines a region tree for all logical regions using that index space
  - Can move from one to the other with:

```
IndexPartition ipart = ...;
LogicalPartition lpart = runtime->get_logical_partition(ctx, lr_root,
                                                        ipart);
assert(ipart == lpart->get_index_partition());
```

- Two APIs for specifying partitions:
  - Coloring, DomainColoring – works now, but has limits
  - Region algebra – planned new approach, coming soon

Partitioning Example (current)

```
// instantiate Coloring data structures (STL maps and sets)
Coloring c_pvtvsshr, c_pvt, c_shr, c_ghost, c_wires;

// manually iterate over the elements in the nodes index space
for(IndexIterator it(is_nodes); it.has_next(); it++
IndexIterator it(is_nodes);
while(it.has_next()) {
  ptr_t n = it.next();

  // some thinking and probably accessing of fields here...
  int owner = ...;  std::set<int> neighbors = ...;

 // add pointer to appropriate sets...
 if(neighbors.empty()) {
    c_pvt[owner].points.insert(n);
    c_pvtvsshr[COLOR_PVT].points.insert(n);
  } else { ... }
}

// now create the index space tree
IndexPartition ip_pvtvsshr = runtime->create_index_partition(ctx, is_nodes, c_pvtvsshr);
IndexSpace is_allpvt = runtime->get_index_subspace(ctx, ip_pvtvsshr, COLOR_  Partition
p_pvt = partition(N_allprivate.part_num);
IndexPartition ip_pvt = runtime->create_index_partition(ctx, is_allpvt, c_pvt);
...
```

Limitations of Current Partitioning API

- Verbosity
  - Direct manipulation of coloring for each element in index space
- Redundancy
  - Common for several partitions to be "related" to each other
  - Each must be specified independently (but consistently!)
- Serial, not distributed
  - Coloring is an STL data structure
  - Not something the Legion runtime understands
  - Not something some code generators will understand either

New Partitioning API (work in progress)

- Eliminates Coloring objects

- Operations to calculate "colorings" become runtime operations
  - Still initiated by application
  - Distributed/deferred by runtime

- Operations divided into three classes:
  - Index-based partitions – based only on properties of the index space
  - Field-based partitions – define partition based on contents of a field
  - Dependent partitions – define partitions based on other partitions

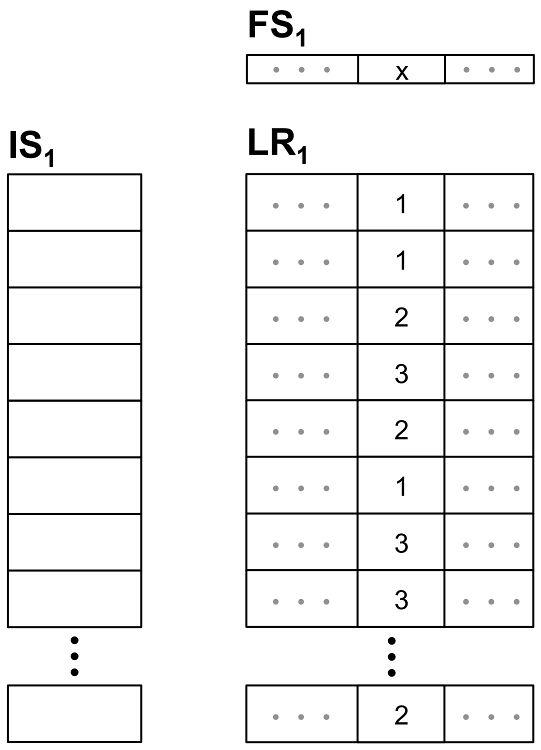
Index-based Partitioning

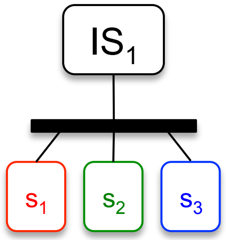
- Unstructured index spaces
  - Effectively sharding
  - Useful for computations with no inter-element communication
  - Useful for providing initial seed for smarter partitioning decisions

- Structured index spaces
  - Blocking
  - Arbitrary sub-rectangles

Field-based Partitioning

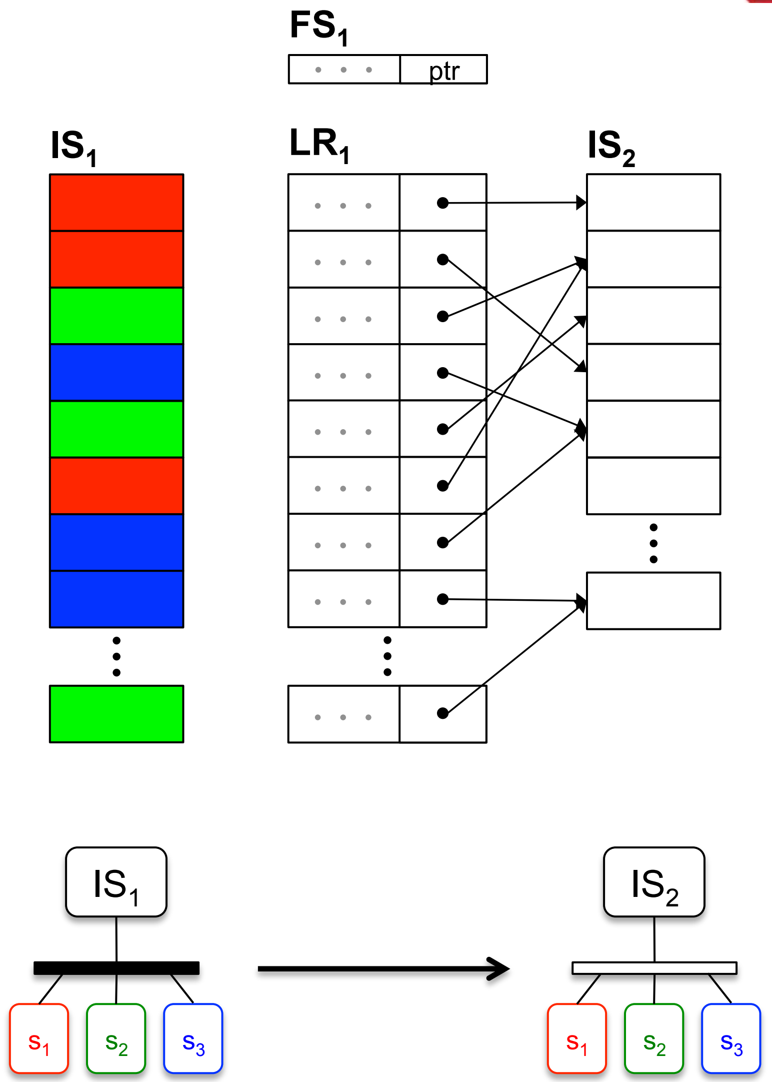
- Use a field's content as the "color" of an element
  - Allows colors to be computed in parallel/distributed fashion
  - Like a "GROUP BY" in SQL
- Partitions are disjoint by definition
  - No efficient way to do "multi-coloring"
- Only raw field value for now





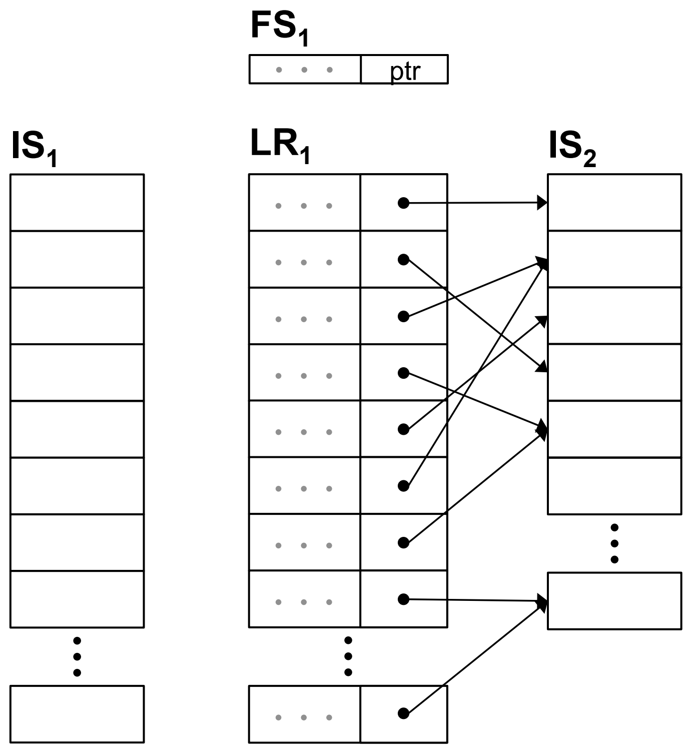
Dependent Partitioning

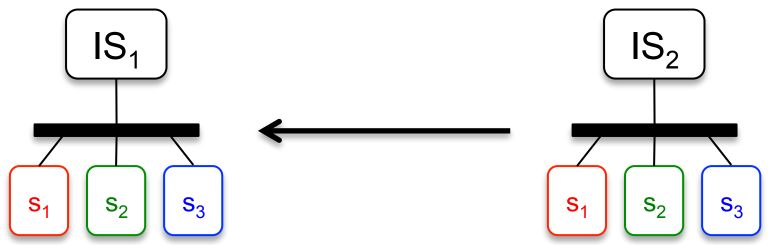
- Set operations
  - Union, intersection, subtraction
  - Partitions: per-element or reduction
- “Join” operations
  - Can be used when field holds pointer to another index space
  - image finds subspaces reachable from an index space or partition



Dependent Partitioning

- Set operations
  - Union, intersection, subtraction
  - Partitions: per-element or reduction
- "Join" operations
  - Can be used when field holds pointer to another index space
  - image finds subspaces reachable from an index space or partition
  - preimage finds subspace that can reach the space/partition





Partitioning Example (improved)

```
task simulate_circuit(Region[Node] N, Region[Wire] W)
{
  parmetis(N, W); // uses index-based partition internally

  // "independent" partition from parmetis' "coloring"
  Partition p_nodes = partition(N.part_num);

  // wires partitioned by ownership of "in" node
  Partition p_wires = preimage(p_nodes, W.in_node);

  // ghost nodes are unowned connected to our wires
  Partition p_ghost = image(p_wires, W.out_node) - p_nodes;

  // shared nodes are those that are ghost for somebody
  Region[Node] N_allshared = union_reduce(p_ghost);

  // private are the others
  Region[Node] N_allprivate = N - N_allshared;

  // private and shared for each circuit piece
  Partition p_pvt = partition(N_allprivate.part_num);
  Partition p_shr = partition(N_allshared.part_num);

  ...
}
```