Overview

Legion & Regent

- *Legion* is
  - a C++ runtime
  - a programming model

- *Regent* is a programming language
  - For the Legion programming model
  - Current implementation is embedded in Lua
  - Has an optimizing compiler
- The bootcamp will focus on Regent

Regent/Legion Design Goals

- Sequential semantics
  - The better to understand what you write
  - Parallelism is extracted automatically
- Throughput-oriented
  - The latency of a single thread/process is (mostly) irrelevant
  - The overall time is what matters
- Runtime decision making
  - Because machines are unpredictable/dynamic

Throughput-Oriented

- Keep the machine busy
- How? Ideally,
  - Every core has a queue of independent work to do
  - Every memory unit has a queue of transfers to do
  - At all times

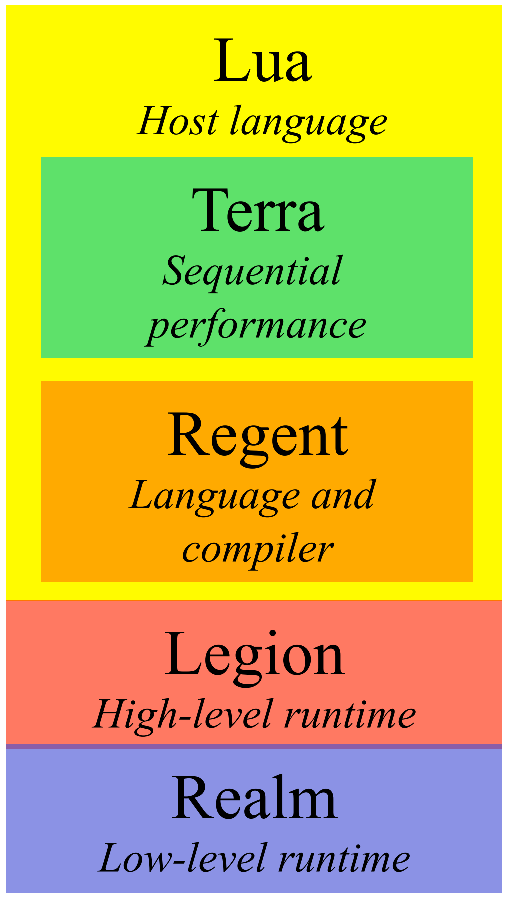
Consequences

- Highly asynchronous
  - Minimize synchronization
  - Esp. global synchronization

- Sequential semantics but support for parallelism

- Emphasis on describing the structure of data
  - Later

Regent Stack



Regent in Lua

- Embedded in Lua
  - Popular scripting language in the graphics community
- Excellent interoperation with C
  - And with other languages
- Python-ish syntax
  - For both Lua and Regent

- Examples Overview/1.rg & 2.rg

- To run:
  - ssh –l USER bootcamp.regent-lang.org
  - cd Bootcamp/Overview
  - qsub r1.sh

Tasks

Tasks

- Tasks are Regent's unit of parallel execution
  - Distinguished functions that can be executed asynchronously
- No preemption
  - Tasks run until they block or terminate
  - And ideally they don't block …

Blocking

- *Blocking* means a task cannot continue
  - So the task stops running
- Blocking does not prevent independent work from being done
  - If the processor has something else to do
  - Does prevent the task from continuing and launching more tasks
- Avoid blocking.

Subtasks

- Tasks can call subtasks
  - Nested parallelism
- Terminology: *parent* and *child* tasks

Example

```
task tester(sum: int64)
...
end

task main()
    var sum: int64 = summer(10)
    sum = tester(sum)
    c.printf("The answer is: %d\n",sum)
end
```

If a parent task inspects the result of a child task, the parent task blocks pending completion of the child task.

- Examples Tasks/1.rg & 2.rg

- Reminder:

  cd Bootcamp/Tasks

  qsub r1.sh

Legion Prof

Legion Prof

- A tool for showing performance timeline
  - Each processor is a timeline
  - Each operation is a time interval
  - Different kinds of operations have different colors

- White space = idle time

Example 1: Legion Prof

```
cd Bootcamp/Tasks
qsub rp1.sh
make prof
```

http://bootcamp.regent-lang.org/~USER/prof1

Example 2: Legion Prof

cd Bootcamp/Tasks
qsub rp2.sh
make prof

http://bootcamp.regent-lang.org/~USER/prof2

Mapping

- How does Regent/Legion decide on which processor to run tasks?

- This decision is under the *mapper's* control

- Here we are using the default mapper
  - Passes out tasks to which CPU on a node is not busy
  - Programmers can write their own mappers
  - More on mapping later

Parallelism

Example Tasks/3.rg

- "for all" style parallelism

- Note the order of completion of the tasks
  - main() finishes first (or almost first)!
  - All subtasks managed by the runtime system
  - Subtasks execute in non-deterministic order

- How?
  - Regent notices that the tasks are *independent*
  - No task depends on another task for its inputs

Runtime Dependence Analysis

- Example Tasks/4.rg is more involved
  - Positive tasks (print a positive integer)
  - Negative tasks (print a negative integer)

- Some tasks are dependent
  - The task for -5 depends on the task for 5
  - Note loop in main() does *not* block on the value of j!

- Some are independent
  - Positive tasks are independent of each other
  - Negative tasks are independent of each other

Legion Spy

Legion Spy

- A tool for showing ordering dependencies
- Very useful for figuring out why things are not running in parallel

Example Tasks/4.rg: Legion Spy

```
cd Bootcamp/Tasks
qsub rs4.sh
make spy
```

http://bootcamp.regent-lang.org/~USER/spy4.pdf

Workflow

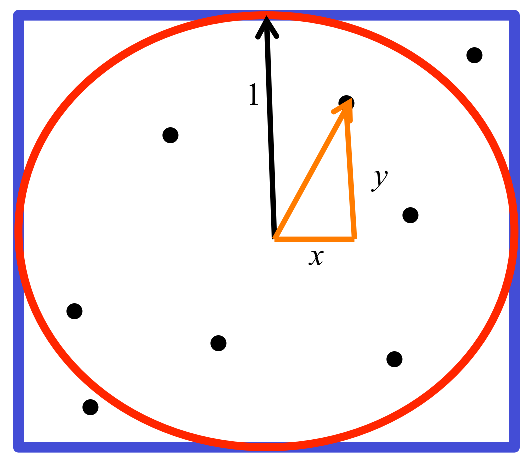
- Use Legion Prof to find idle time
  - white space
- Use Legion Spy to examine tasks that are delayed
  - What are they waiting for?!

Exercise 1

Computing the Area of a Unit Circle

- A Monte Carlo simulation to compute the area of a unit circle inscribed in a square

- Throw darts
  - Fraction of darts landing in the circle = ratio of circle's area to square's area



Computing the Area of a Unit Circle

- Example Pi/1.rg
  - Slow!
  - Why?

Exercise 1

- Modify Pi/1.rg
  - Edit x1.rg
  - make multiple trials per subtask

- Use
  - 4 subtasks
  - 2500 trials per subtask

- Produce both prof and spy output
  - See Makefile

Terra

Leaf Tasks

- *Leaf tasks* call no other tasks
  - The "leaves" of the task tree
- Leaf tasks are sequential programs
  - And generally where the heavy compute will be
- Thus, leaf tasks should be optimized for latency, not throughput
  - Want them to finish as fast as possible!

Terra

- Terra is a low-level, typed language embedded in Lua
- Designed to be like C
  - And to compile to similarly efficient code
- Also supports vector intrinsics
  - Not illustrated today

Terra Example

- Terra/1.rg converts the *hits* task in Terra/x1.rg to a Terra function

- Trivial in this example
  - Just change ”task” to “terra”
  - Marginally faster
    - On average …

Considerations in Writing Regent Programs

- The granularity of tasks must be sufficient
  - Don't write very short running tasks
- Don't block in tasks that launch many subtasks
- Terra is an option for heavy sequential computations

Structured Regions

Regions

- A region is a (typed) collection

- Regions are the cross product of
  - An *index space*
  - A *field space*

StructuredRegions/1.rg

	Bit
0	false
1	false
2	false
3	false
4	false
5	true
6	true
7	true
8	true
9	false

Discussion

- Regions are *the* way to organize large data collections in Regent
- Regions can be
  - Structured (e.g., like arrays)
  - Unstructured (e.g., pointer data structures)
- Any number of fields
- Built-in support for 1D, 2D and 3D index spaces

Privileges

- A task that takes region arguments must
  - Declare its *privileges* on the region
  - Reads, Writes, Reduces
- The task may only perform operations for which it has privileges
  - Including any subtasks it calls

- Example StructuredRegions/2.rg
- Example StructuredRegions/3.rg

Reduction Privileges

- StructuredRegions/4.rg
  - A sequence of tasks that increment elements of a region
  - With Read/Write privileges

- StructuredRegions/5.rg
  - 4.rg but with Reduction privileges

- Note: Reductions can create additional copies
  - To get more parallelism
  - Under mapper control
  - Not always preferred to Read/Write privileges

Partitioning

Partitioning

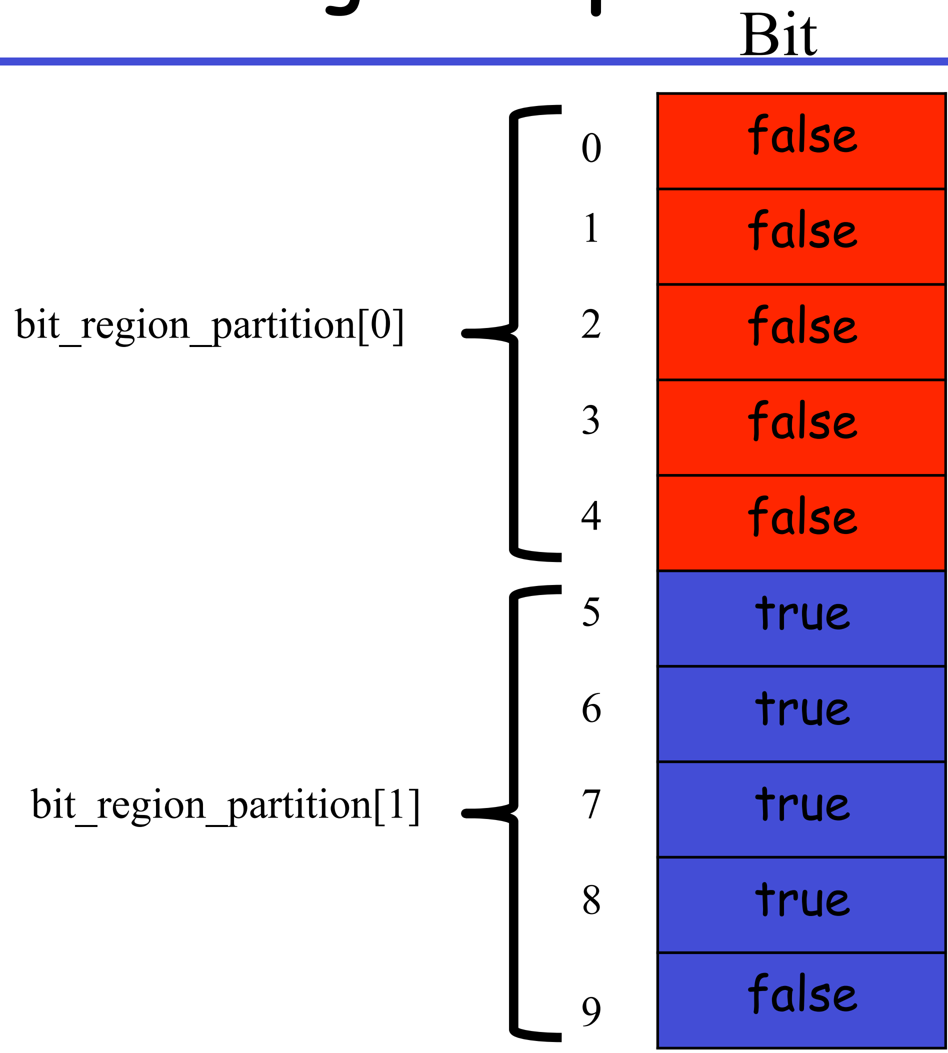
- To enable parallelism on a region, *partition* it into smaller pieces
  - And then run a task on each piece

- Legion/Regent have a rich set of partitioning primitives

Partitioning Example

	Bit
0	false
1	false
2	false
3	false
4	false
5	true
6	true
7	true
8	true
9	false

Partitioning Example



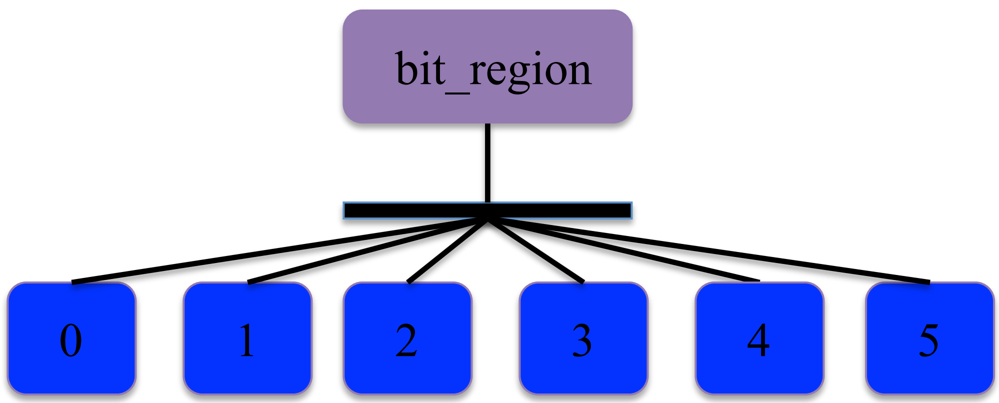
Equal Partitions

- One commonly used primitive is to split a region into a number of (nearly) equal size subregions
- Partitioning/1.rg
- Partitioning/2.rg

Discussion

- Partitioning does not create copies
  - It names subsets of the data
- Partitioning does not remove the parent region
  - It still exists and can be used
- Regions and partitions are first-class values
  - Can be created, destroyed, stored in data structures, passed to and returned from tasks

Region Trees



More Discussion

- The same data can be partitioned multiple ways
  - Again, these are just names for subsets

- Subregions can themselves be partitioned

Dependence Analysis

- Regent uses tasks' region arguments to compute which tasks can run in parallel
  - What region is being accessed
    - Does it overlap with another region that is in use?
  - What field is being accessed
    - If a task is using an overlapping region, is it using the same field?
  - What are the privileges?
    - If two tasks are accessing the same field, are they both reading or both reducing?

A Crucial Fact

- Regent analyzes *sibling* tasks
  - Tasks launched directly by the same parent task

- Theorem: Analyzing dependencies between sibling tasks is sufficient to guarantee sequential semantics

- Never check for dependencies otherwise
  - Crucial to the overall design of Regent

Consequences

- Dependence analysis is a source of runtime overhead
- Can be reduced by reducing the number of sibling tasks
  - Group some tasks into subtasks
- But beware!
  - This may also reduce the available parallelism
- Partitioning/3.rg

Partitioning/3.rg

- Note that passing a region to a task does not mean the data is copied to where that task runs
  - C.f., launcher task must name the parent region for type checking reasons

- If the task doesn't touch a region/field, that data doesn't need to move

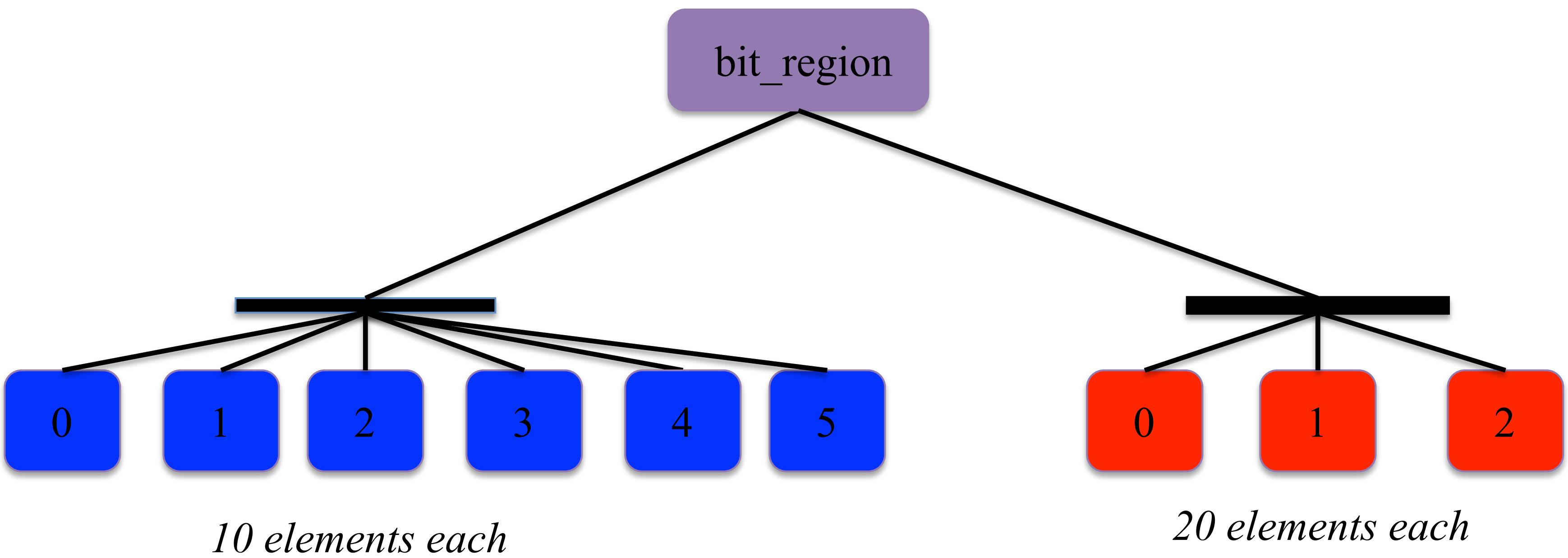
Fills

- A better way to initialize regions is to use *fill* operations

fill(region.field, value)

- Partitioning/4.rg

Multiple Partitions



Discussion

- Different views onto the same data
- Again, can have multiple views in use at the same time
- Regent will figure out the data dependencies

Exercise 2

- Modify Partitioning/4.rg to
- Have two partitions of bit_region
  - One with 3 subregions of size 20
  - One with 6 subregions of size 10
- In a loop, alternately launch subtasks on one partition and then the other
- Edit x2.rg

Aliased Partitions

- So far all of our examples have been *disjoint partitions*
- It is also possible for partitions to be *aliased*
  - The subregions overlap
- Partitioning/5.rg

Partitioning Summary

- Significant Regent applications have interesting region trees
  - Multiple views
  - Aliased partitions
  - Multiple levels of nesting

- And complex task dependencies
  - Subregions, fields, privileges, coherence

- Regions express locality
  - Data that will be used together
  - An example of a "local address space" design
    - Tasks can only access their region arguments

Image Blur

Index Notation

- First example with a 2D region

- Rect2d type
  - 2D rectangle
  - To construct: rect2d { lo, hi }
  - Note lo and hi are 2D points!
  - Fields: r.lo, r.hi
  - Operation: r.lo + {1,1}, r.hi – {1,1}

- The following works (modulo bounds):

```
for x in r do
   r[x + {1,1}]
```

Blur

- Compute a Gaussian blur of an image
- Edit Blur/blur.rg
  - Search for TODO
  - … in two separate places …
  - Test with qsub rpblur.sh
- Solution is in blur_solution.rg
  - Also scripts for running the solution
  - With and without GPUs

Unstructured Regions

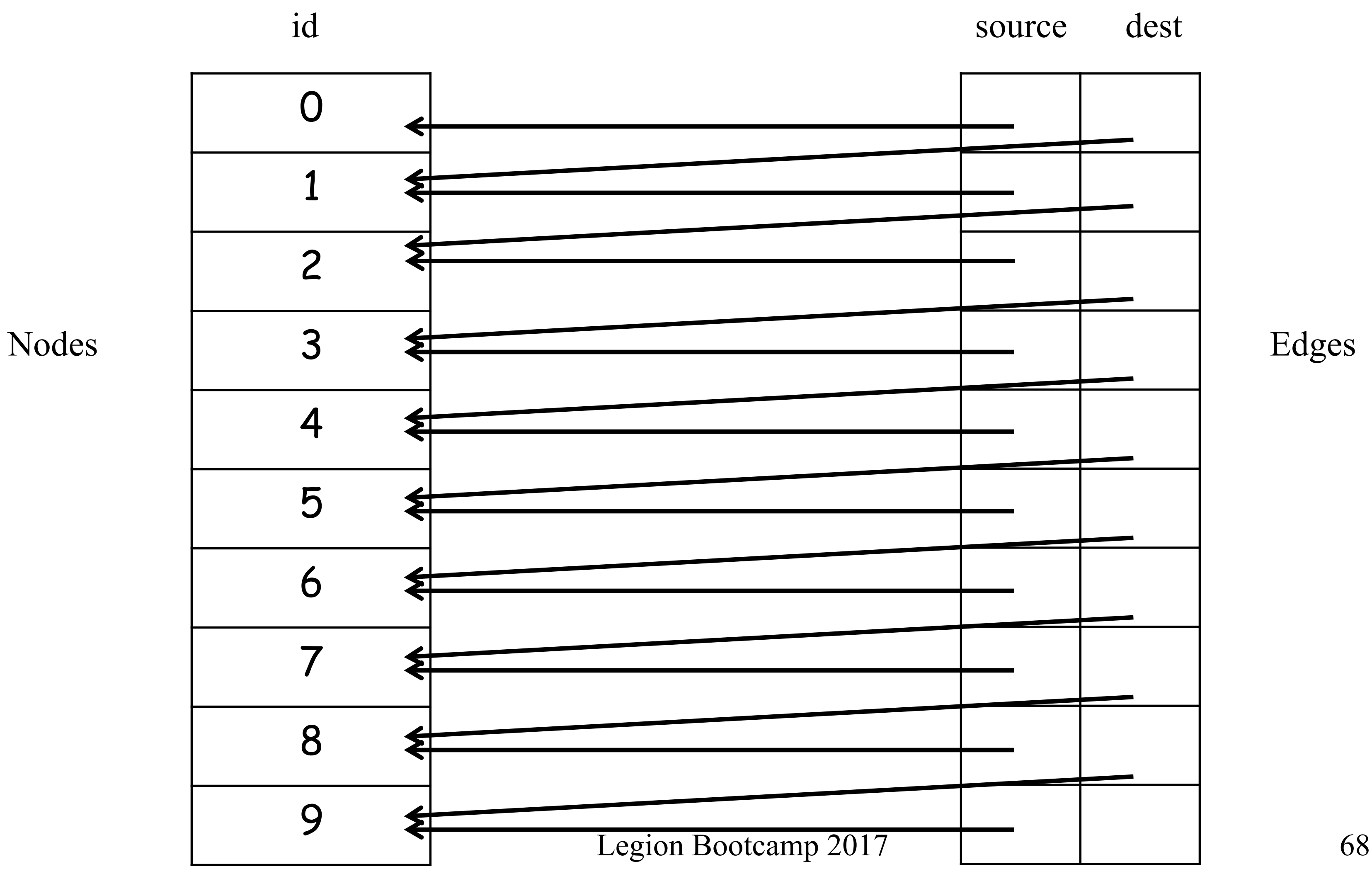
Regions Review

- A region is a (typed) collection
- Regions are the cross product of
  - An *index space*
  - A *field space*
- A *structured region* has a structured index space
  - E.g., int1d, int2d, int3d

new(…)

- Unstructured regions have a size
- But initially they have no elements
- Elements are allocated using new(…)
  - Occupies one (as yet) unallocated element of the region

UnstructuredRegions/1.rg and 2.rg



Partitioning By Field

- A field can be used as a coloring
- Write elements of the color space into the field f
  - Using an arbitrary computation
- Then call partition(region.f, colors)
  - UnstructureRegions/3.rg

Dependent Partitioning

Partitioning, Revisited

- Why do we want to partition data?
  - For parallelism
  - We will launch many tasks over many subregions

- A problem
  - We often need to partition multiple data structures in a consistent way
  - E.g., given that we have partitioned the nodes a particular way, that will dictate the desired partitioning of the edges

Dependent Partitioning

- Distinguish two kinds of partitions

- *Independent partitions*
  - Computed from the parent region, using, e.g.,
    - partition(equals, …)

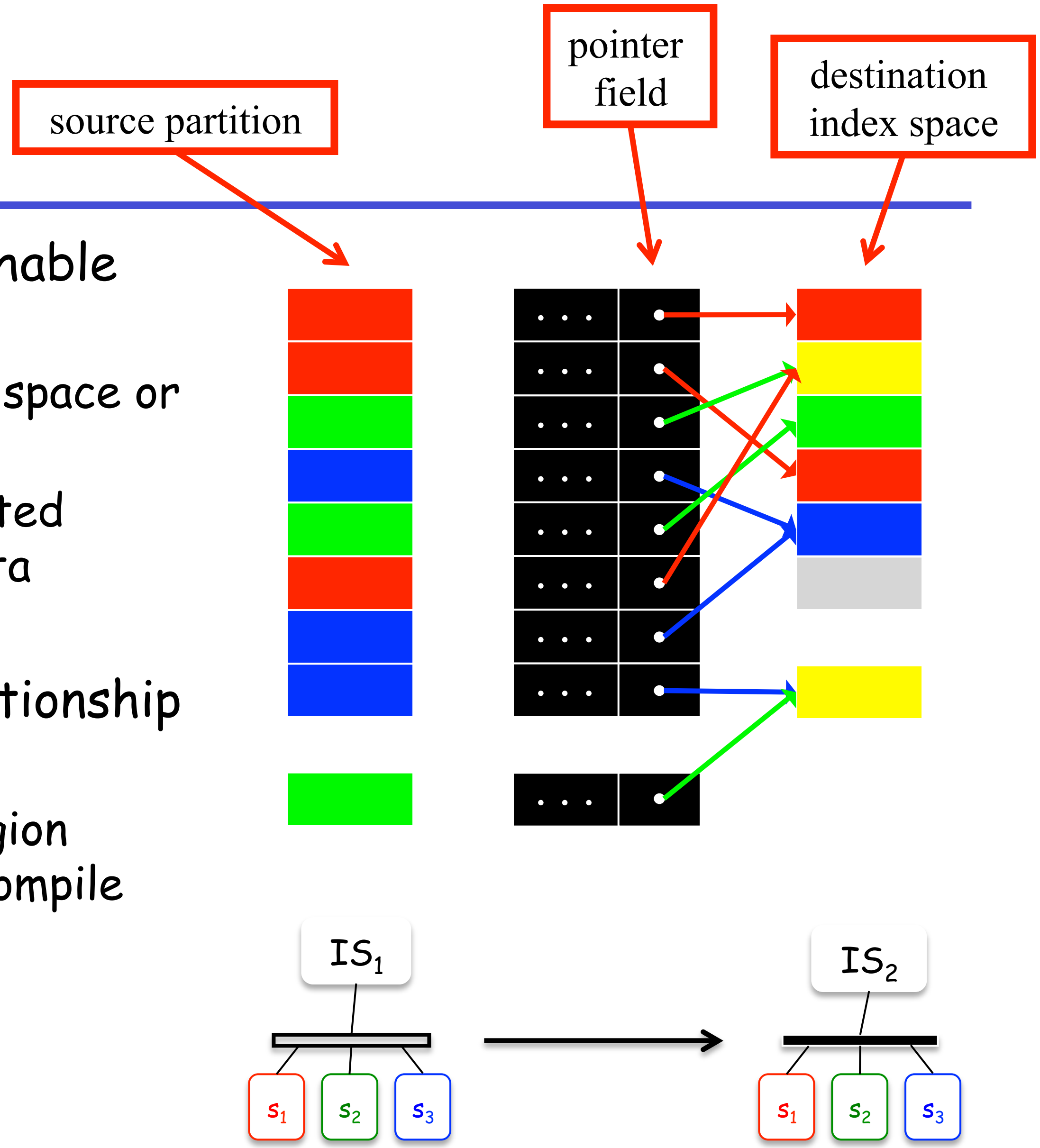
- *Dependent partitions*
  - Computed using another partition

Dependent Partitioning Operations

- Image
  - Use the image of a field in a partition to define a new partition
- Preimage
  - Use the pre-image of a field in a partition …
- Set operations
  - Form new partitions using the intersection, union, and set difference of other partitions

Image

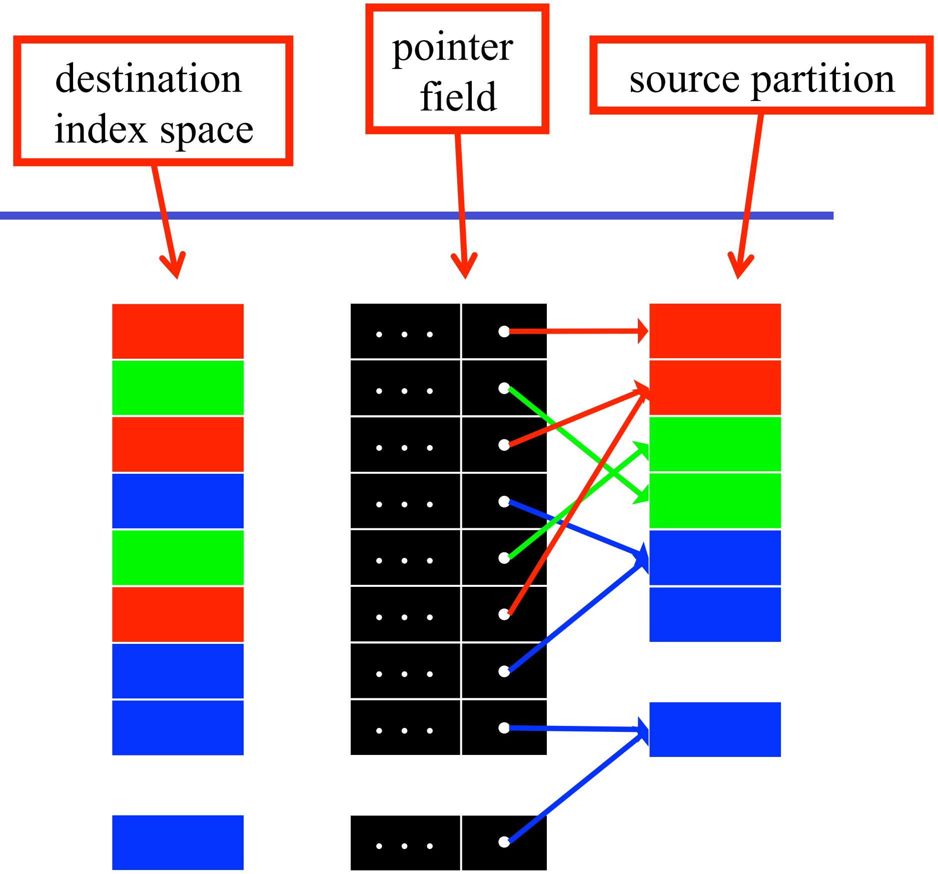
- Computes elements reachable via a field lookup
  - Can be applied to index space or another partition
  - Computation is distributed based on location of data
- Regent understands relationship between partitions
  - Can check safety of region relation assertions at compile time

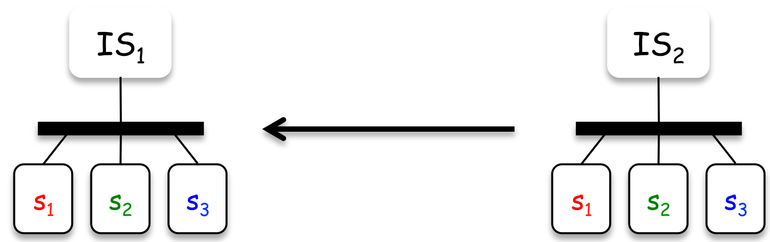


Preimage

- Inverse of image
  - Computes elements that reach a given subspace
  - Preserves disjointness

- Multiple images/preimages can be combined
  - Can capture complex task access patterns





DependentPartitioning/1.rg

- Partition the nodes
  - Equal partitioning

- Then partition the edges
  - Preimage of the source node of each edge

- For each node subregion r, form a subregion of those edges where the source node is in r

DependentPartitioning/2.rg

- Partition the edges
  - Equal partitioning
- Then partition the nodes
  - Image of the source node of each edge
- For each edge subregion r, form a subregion of those nodes that are source nodes in r

Discussion

- Note that these two examples compute (almost) the same partition
- Can derive the node partition from the edges, or vice versa

Exercise

- What would the example look like if we partitioned based on the destination node?

- Let's find out …
  - Modify 2.rg to partition using the destination node
  - Code is in DependentPartitioning/x3.rg

Set Operations: Set Difference

- Partition the edges
  - Equal partition
- Compute the source and destination node partitions of the previous two examples
- The final node partition is the set difference
  - What does this compute?
  - Examples DepedendentPartitioning/4.rg & 5.rg

Set Operations: Set Intersection

- Partition the edges
  - Equal partition

- Compute the source & destination node partitions

- Final node partition is the intersection
  - What does this compute?
  - Example DependentPartitioning/6.rg

DependentPartitioning/7.rg

- Same as the last example

- Once the final node partition is computed, compute a partition of the edges such that each edge subregion has only the edges connecting the nodes in the corresponding node subregion

Some Comments on Type Checking

TypeChecking/1.rg

- Pointers point into a particular region
  - And this is part of the pointer's type

- Partitioning can change which region(s) a pointer points to
  - May lead to typechecking issues, depending on which region you want to use for an operation

TypeChecking/2.rg

- The right way to fix type issues is to use type casts
- Very analogous to downcasting from a more general object type to a more specific object type in an object-oriented language
- But, this solution does not currently work!
  - Casting of region types not yet implemented

TypeChecking/3.rg

- The fix/workaround is to use wild in field space arguments when allocating regions
- Wild effectively turns off typechecking for those region arguments.

Page Rank

The Algorithm

- The page rank algorithm computes an iterative solution to the following equation, where
  - PR(p) is the probability that page p is visited
  - N is the number of pages
  - L(p) is the number of outgoing links from p
  - d is a "damping factor" between 0 and 1

$$PR(p) = \frac{1-d}{N} + d \sum_{p' \in M(p)} \frac{PR(p')}{L(p')}$$

Exercise

- Modify Pagerank/pagerank.rg

- Play with the partitioning of the graph
- And possibly the permissions (hint!)