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

Example 21

<table>
<thead>
<tr>
<th>id</th>
<th>source</th>
<th>dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Nodes

Edges
Partitioning

• Unstructured regions can be partitioned

• In particular, partitioning into equal size chunks works as expected.

• Example 20

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 $\text{partition}(\text{region}.f, \text{colors})$
  - Example 27
  - Preferred to using explicit coloring objects
Partitioning, Digression

• 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, ... )
    • partition(region.field, ... )

• 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
  - Equivalent to semi-join in relational algebra
  - 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

- Opposite of image - computes elements that reach a given subspace
  - Preserves disjointness

- Multiple images/preimages can be combined
  - can capture complex task access patterns
  - Limitation: no transitive reachability

Example 21

- 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 \( r \)
Example 22

• 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 the same partition

• Can derive the node partition from the edges, or vice versa
Example 23

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

• Let’s find out ...

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 24 & 25
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 26

Example 28

• 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
Examples 29

- 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

Example 30

- 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
Example 31

• The fix/workaround is to use wild in field space arguments when allocating regions

• Wild effectively turns off typechecking for those region arguments.

Backing Up ...

• Regent’s partitioning mechanisms are very different from other languages

• What do those other languages provide?
**One Extreme: Simplicity**

- PGAS languages (e.g. X10, UPC, Chapel) generally provide only simple array-based distribution methods
  - e.g. block, cyclic, blockcyclic
- **Pros:**
  - simple for programmer to describe
  - simple for compiler to verify consistency
  - simple for runtime to implement
- **Cons:**
  - no support for irregular (or even semi-regular) data structures
  - no support for irregular partitions of structured data
  - no support for aliased or multiple partitions

**Other Extreme: Expressivity**

- Initial Legion partitioning used general-purpose coloring object for ALL partitioning operations
  - Application able to color each element any way it wants
- **Pros:**
  - support for arbitrary irregularity in data and/or partitioning
  - support for aliased partitions, multiple partitions
- **Cons:**
  - significant programmer effort to describe even simple partitions
  - no ability for compiler to check that related regions are partitioned consistently
  - high runtime overhead for computing and querying partitions
  - manipulation of coloring was serial, limited to single node
Dependent Partitioning

- A carefully chosen middle ground between these two extremes
- Supports both structured and unstructured domains
- Allows arbitrary independent partitions to be computed by the application
  - But uses field data to capture intent rather than a coloring
  - Index-based partitions cover PGAS-like simple cases
- Provides an analyzable set of operations to compute dependent partitions from other partitions
  - Based on reachability and/or set operations
  - Consistency of dependent partitions can be verified at compile time
- And can be executed in parallel

Programmer Productivity

- Lines of code for computation of dependent partitions in Regent applications:

<table>
<thead>
<tr>
<th>Application</th>
<th>Original LOC</th>
<th>Dependent Partitioning LOC</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PENNANT</td>
<td>163</td>
<td>6</td>
<td>96%</td>
</tr>
<tr>
<td>Circuit</td>
<td>159</td>
<td>8</td>
<td>95%</td>
</tr>
<tr>
<td>MiniAero</td>
<td>51</td>
<td>7</td>
<td>86%</td>
</tr>
</tbody>
</table>

- Not a perfect metric
  - Take with however much salt you like...
Summary

• The built-in partitioning operations are
  - Expressive
  - Can execute in parallel
  - Can be analyzed by the Regent implementation

• Except for explicit coloring objects
  - Inherently not parallel
  - Will eventually go away