The Legion Programming Model

ATPESC 2018

Wonchan Lee
Stanford University, Los Alamos National Laboratory

Q Center, St. Charles, IL (USA)
August 2, 2018
• The Legion project is joint work between Stanford, Los Alamos National Lab, NVIDIA, and SLAC.

• Funding has come from many sources, but particularly the DOE and the leadership class facilities.

• This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy’s Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation’s exascale computing imperative.
What Do You Need Today?

- A laptop
  - With access to the ATPESC WIFI (Q Basic)

- A shell & ssh

- Login credentials
  - You should already have received this
  - But we can also give you credentials during the hands-on session

- Example programs are also at https://tinyurl.com/legion-atpesc18
Overview
Legion & Regent

• *Legion* is a
  – C++ runtime
  – Programming model

• *Regent* is a programming language
  – For the Legion programming model
  – Current implementation is embedded in Lua
  – Has an optimizing compiler

• This tutorial focuses on Regent
• 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
• 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

- **Lua**
  - Host language

- **Terra**
  - Sequential performance

- **Regent**
  - Language and compiler

- **Legion**
  - High-level runtime

- **Realm**
  - Low-level runtime
Regent in Lua

• Embedded in Lua
  – Popular scripting language in the graphics community

• Excellent interoperation with C
  – And with other languages

• Simple syntax
  – For both Lua and Regent
• Examples Overview/1.rg & 2.rg

• To run:
  – ssh -l USER atpesc18.regent-lang.org
  – cd atpesc18/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** 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
task summer(num : int64) : int64 ... end

task tester(sum : int64) ... end

task main()
  var sum : int64 = summer(10)
  sum = tester(sum)
  c.printf("The answer is: %ld\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 atpesc18/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 atpesc18/Tasks
qsub rp1.sh
make prof

http://atpesc18.regent-lang.org/~USER/prof
Example 2: Legion Prof

cd atpesc18/Tasks
qsub rp2.sh
make prof

http://atpesc18.regent-lang.org/~USER/prof.1
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 CPUs on a node in a round-robin fashion
- Programmers can write their own mappers
- More on mapping later
Parallelism
“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
• 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
• 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 atpesc18/Tasks
qsub rs4.sh
make spy

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
Regions
Regions

• A region is a (typed) collection

• Regions are the cross product of
  – An index space
  – A field space
<table>
<thead>
<tr>
<th>bit</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>
• Regions are *the* way to organize large data collections in Regent

• Regions can be
  – Dense (e.g., like arrays)
  – Sparse (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 Regions/2.rg

• Example Regions/3.rg
Reduction Privileges

• Regions/4.rg
  – A sequence of tasks that increment elements of a region
  – With Read/Write privileges

• Regions/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
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_region_partition[0]**
  - 0: false
  - 1: false
  - 2: false
  - 3: false
  - 4: false
  - 5: true

- **bit_region_partition[1]**
  - 0: false
  - 1: false
  - 2: false
  - 3: false
  - 4: false
  - 5: true
  - 6: true
  - 7: true
  - 8: true
  - 9: true
One commonly used primitive is to split a region into a number of (nearly) equal size subregions

- Partitioning/1.rg
- Partitioning/2.rg
• 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

bit_region

bit_region_partition

0 1 2 3 4 5
More Discussion

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

- Subregions can themselves be partitioned
• 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?
• 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
• 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
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

\[
\text{fill}(\text{region.field}, \text{value})
\]

• Partitioning/4.rg
Multiple Partitions

bit_region

10 elements each

20 elements each
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/x2.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
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
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

• Partition by field
  – Group elements by the value of a field

• 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 others
Partitioning By Field

- Write elements of the color space into the field $f$
  - Using an arbitrary computation

- Then call $\text{partition}(\text{region}.f, \text{colors})$
  - DependentPartitioning/0.rg
Partitioning By Field

- Write elements of the color space into the field $f$
  - Using an arbitrary computation

- Then call `partition(region.f, colors)`
  - DependentPartitioning/0.rg

```
0 → 1 → 2 → 3 → 4 → 5 → ...
```

```
```

```
0  4  ...  1  5  ...  2  ...  3  ...
```
• Computes elements reachable via a field lookup
  – Computation is distributed based on location of data

• Regent understands relationship between partitions
  – Can check safety of region relation assertions at compile time
• 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 \( r \)
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
• 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 $r$
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 1.rg to partition using the destination node
  – Code is in DependentPartitioning/x3.rg
• 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
• 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
• 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
Mapping
Mapping

• Mapping is the process of assigning resources to Regent/Legion programs

• Conceptually
  – Assign a processor to each task
    • The task will execute in its entirety on that processor
  – Assign a memory to each region argument

• And many other things!
Understanding Mappers

• Mapping is an API
  – A set of callbacks

• Each is called at a particular point in a task’s lifetime
  – To write mappers, need to know this sequence of stages
The Legion Mapping API

• At the Legion level, mapping is an API
  – A set of callbacks
  – Each is called at a particular point in a task’s lifetime
  – To write mappers, need to know this sequence of stages

• Regent has a mapping DSL
  – Concise, easy to use
  – Compiles to the Legion mapping API
  – Currently supports only static mappings
High-Level Overview

• An instance of the Legion runtime runs on every node

• When a task is launched the local runtime
  – Makes mapper calls to pick a processor for the task
  – Makes mapper calls to pick memories for the region arguments
  – … and other mapper calls as well …
Conclusions
Conclusions

• Legion/Regent is a task-based parallel programming system

• Advantages
  – Easy to exploit multiple levels of parallelism in a uniform manner
  – Novel and rich partitioning sublanguage
  – Separate machine mapping

• Good/great performance and portability!
Thank you!
Backup Slides
Image Blur
• 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] = r[x + {1,1}] + ...
  ```
• 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
Page Rank
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
  - Can you switch from a page-based partitioning to a link-based partitioning?

- And possibly the permissions
  - See “TODO”
Mapping
New Concepts

- There are a number of concepts at the mapping level that don’t exist in Regent
  - Machine models
  - Variants
  - Physical Instances
- More on this later . . .
To pick concrete processors & memories, the runtime must know:

- How many processors/memories there are
  - And of what kinds

- And where the processors/memories are
  - At least relative to each other
Machine Model

- Processors
  - LOC
  - TOC
  - PROC_SET
  - UTILITY
  - IO

- Memories
  - GLOBAL
  - SYSTEM
  - RDMA
  - FRAME_BUFFER
  - ZERO_COPY
  - DISK
  - HDF5
Affinities

- Processor -> Memory
  - Which memories are attached to a processor

- Memory -> Memory
  - Which memories have channels between them

- Memory -> Processor
  - All processors attached to a memory

- Affinities are provided as a list of \((\text{proc,mem})\) and \((\text{mem,mem})\) pairs
Task Variants

• A task can have multiple *variants*
  – Different implementations of the same task
  – Multiple variants can be registered with the runtime
  – Variants can have associated *constraints*

• Examples
  – A variant for LOC
  – Another variant for TOC
  – Variants for different data layouts
Physical Instances

- A region is a logical name for data
- A physical instance is a copy of that data
  - For some set of fields
- There can be 0, 1 or many physical instances of a specific field of a region at any time
Physical Instances

- Can be *valid* or *invalid*
  - Is the data current or not?

- Live in a specific memory

- Have a specific layout
  - Column major, row major, blocked, struct-of-arrays, array-of-structs, …

- Are allocated explicitly by the mapper

- Are deallocated by the runtime
  - Garbage collected
• Many physical instances of a region can exist simultaneously
  – Including different versions of the same data

• A task writing version 0 to disk
• A task reading version 5
• A task writing version 6
  – The current version!
• A task scheduled to read version 6
• A task scheduled to write version 7
• A (meta)task scheduled to deallocate version 6
• …
Create Mappers

- Called once on start-up
  - On each node
There are three stages, in order:

- **Select task options**
  - Like it says, choose among some options

- **Slice task**
  - Break up index launches into chunks and distribute
  - Fixes the node of the task

- **Map task**
  - Bind the task to a processor
Controlling Processor Choice in Regent

- Place immediately before a task declaration
  - `__demand(__cuda)`

- Causes both CPU and GPU task variants to be produced

- And the default mapper always prefers to pick a GPU variant if possible
• Tasks can have layout constraints on physical instances
  – “This task requires data in row major order”
  – Multiple instances may satisfy the constraints
The default mapper first checks if there is an existing valid instance for a region requirement
- That satisfies the layout constraints
- And has affinity to the processor

If so, return it

If not, create a new instance
- In system memory (for a CPU mapped task)
- In frame buffer memory (for a GPU mapped task)
Summary

• Mapping
  – Selects processors for tasks
  – Selects memories for physical instances
    • Satisfying region requirements of tasks

• Many options
  – Default mapper does reasonable things
  – But any sufficiently complex program will need some customization