The Legion Programming Model

(and the Regent compiler)

ATPESC – 2017
Programming HPC systems is hard today and will only get harder without new approaches

- Aspects of programming future large-scale systems
  - Focusing on full-system, data-awareness and improved productivity
    - Programming in the small is not the challenge, we must take a broader view
    - Co-designing the full tool chain with applications
    - End-to-end awareness is required to avoid point solutions “non-solutions”
  - Targeting large scale dynamic computation environments
    - Hardware dynamics: frequency scaling, dark silicon, adaptive routing, …
    - Software dynamics: system services, in-situ/co-resident services, …
    - Application dynamics: multiscale and multiphysics

- Programming model goals (What must we deliver?)
  - High performance – we must be fast
  - Performance portability – across many kinds of machines over many generations
  - Programmability – sequential semantics, parallel execution
Can we fulfill our programming model goals today?

We can, to some degree…
… but at great cost: Programmer Pain

Do you want to schedule this graph?
(High Performance)

Do you want to re-schedule this graph for every new machine?
(Performance Portability)

Do you want to be responsible for generating this graph?
(Programmability)

Today: programmer’s responsibility

Tomorrow: programming system’s responsibility

Task graph for one time step on one node…
… of a mini-app
We need the right programming abstractions to achieve our goals

- Today’s programming environments:
- Are using the wrong abstractions...
- Focus on control flow, parallelism and have low-level data abstractions (i.e. no data model)

```
AsyncRecv(X);
DoWork(Y);
Sync();
F(X);
```

- How much work should I do?
- Is this performance portable?
- When does forward progress really occur?
- What if I have more work and data movement happening in DoWork?
- What resources are in use? Where is the data? Who is using it and how?
- Is this modular?
- What if there is a fault?

Concept from: Mike Bauer’s Thesis (Stanford),
Legion: Programming Distributed Heterogeneous Architectures with Logical Regions
Our approach to meeting our programming model goals is different

- Legion Programs
- Legion Mappers
  - Extraction of parallelism
  - Task scheduling and Latency hiding
  - Management of data transfers
  - Data-Dependent Behavior
Legion: Separation of Concerns

**Tasks** (execution model)
- Describe parallel execution elements and algorithmic operations with sequential semantics, out-of-order execution

**Regions** (data model)
- Describe decomposition of computational domain.
  - Privileges (read-write, read-only, reduce)
  - Coherence (exclusive, atomic)

**Mapper**
- Describes how tasks and regions should be mapped to the target architecture
- Mapper allows architecture-specific optimization without effecting the correctness of the task or domain descriptions

\[ (\text{int } i) \{ \text{rho}(i) = \ldots \} \]
The Legion programming model: enabling the runtime to manage the complexity of a dynamic environment

- **Legion Program**
  - Machine-Independent Specification
  - Defines application correctness
  - Sequential semantics

- **Legion**

- **Legion Mapper**
  - Programmatic interface for performing machine-specific and app-specific mapping
  - Only impacts performance
  - Yesterday: manual mapping
  - Today: programmatic mapping
  - Tomorrow: generated mappers
Data Model – Logical Regions

- Unbounded set of rows (index space)
- Bounded set of columns (fields)
- Can be partitioned (disjoint or aliased)
- **Tasks operate on regions**
  - Must specify which fields and how they “use” them (fields and privileges: read, write, read+write, exclusive, etc.)
  - Allows fields to be “sliced”
- Tasks launched in program order (execution order relaxed based on dependencies)
  - “out of order” software processor
Our approach shows compelling results with a full application

- Ported production combustion application (S3D)
  - Sufficient complexity to validate our approach – beyond the “Proxy”

- High-level application constructs remain in MPI form… - essentially all the initial setup was kept

- Full right hand side function implemented in Legion including all communication (the full stencil computation and all chemistry calculations)
  - ~2-3X faster than MPI+OpenACC
  - ~7X faster than MPI only

- Enabling new science that is not possible with current approaches
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

• This tutorial 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 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
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

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

• Compare $\Pi/1.rg$ and $\Pi/x1.rg$
  - Identify use of multiple trials per subtask
• Modify $\Pi/x1.rg$ (change “terra hits” to “task hits”)
• Uses
  - 4 subtasks
  - 2500 trials per subtask
• Which is faster? Why?
  - Hint: Use Legion Prof and Legion Spy
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
<table>
<thead>
<tr>
<th>Bit</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>false</td>
</tr>
<tr>
<td>1</td>
<td>false</td>
</tr>
<tr>
<td>2</td>
<td>false</td>
</tr>
<tr>
<td>3</td>
<td>false</td>
</tr>
<tr>
<td>4</td>
<td>false</td>
</tr>
<tr>
<td>5</td>
<td>true</td>
</tr>
<tr>
<td>6</td>
<td>true</td>
</tr>
<tr>
<td>7</td>
<td>true</td>
</tr>
<tr>
<td>8</td>
<td>false</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>
Partitioning Example

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

bit_region_partition[0]

bit_region_partition[1]
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
  - Question: Why does this hold? (Intuitively)

• 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

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/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
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
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