The Legion Programming Model

Alex Aiken

Elliott Slaughter

Stanford University & SLAC

What Do You Need Today?

- A laptop
  - With access to the conference wifi

- A shell & ssh

- Login credentials
  - You should already have received these
  - But we can also give you credentials now

Acknowledgments

- The Legion project is joint work between Stanford, CMU, Los Alamos National Lab, SLAC, and NVIDIA.

- Funding has come from many sources, including NSF and DARPA, but particularly the DOE and the leadership class facilities.

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

- This tutorial focuses 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
- Compositional
  - To enable writing parallel/distributed libraries

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

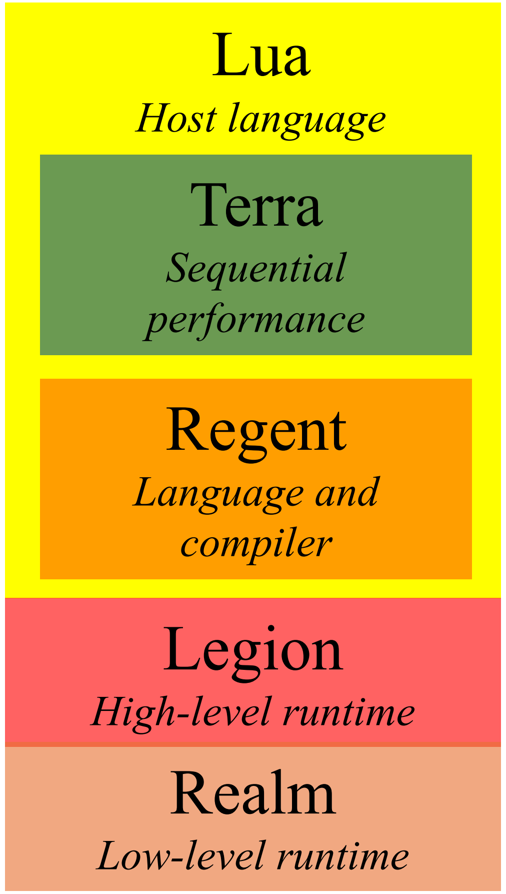
Consequences

- 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 USER@sapling.stanford.edu
  - cd sc24
  - source env.sh
  - cd Overview
  - sbatch r1.sh
  - squeue --me # wait until job completes
  - less slurm-*.log

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)
    format.println("The answer is: {}",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 sc24/Tasks

  sbatch r1.sh

  squeue --me

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 sc24/Tasks
sbatch rp1.sh
squeue --me # wait for job to complete
legion_prof_to_public_html prof1_*.gz
```

Open the URL printed to the terminal

Example 2: Legion Prof

```
cd sc24/Tasks
sbatch rp2.sh
squeue --me # wait for job to complete
legion_prof_to_public_html prof2_*.gz
```

Open the URL printed to the terminal

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 CPUs that are 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

Workflow

- Use Legion Prof to find idle time
  - white space

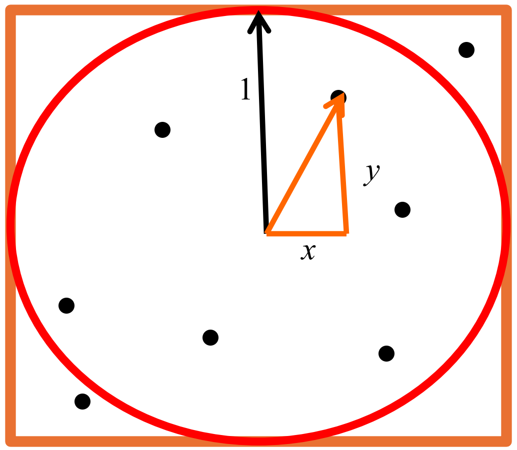
- The profile can also show why a task ran when it did
  - What was the task waiting for?!

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 profiling output
  - Run legion_prof_to_public_html on the resulting profile logs

- Lesson: Task *granularity* is important
  - Tasks that are too short cost more for the runtime system to analyze than to execute!
  - For Legion, an average task granularity of 1ms is recommended

Regions

Regions

- A region is a (typed) collection

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

Regions/1.rg

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

Discussion

- Regions are *the* way to organize large data collections in Regent

- Index spaces can be dense or sparse

- 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 (with a specific reduction operator)

- 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

Partitioning

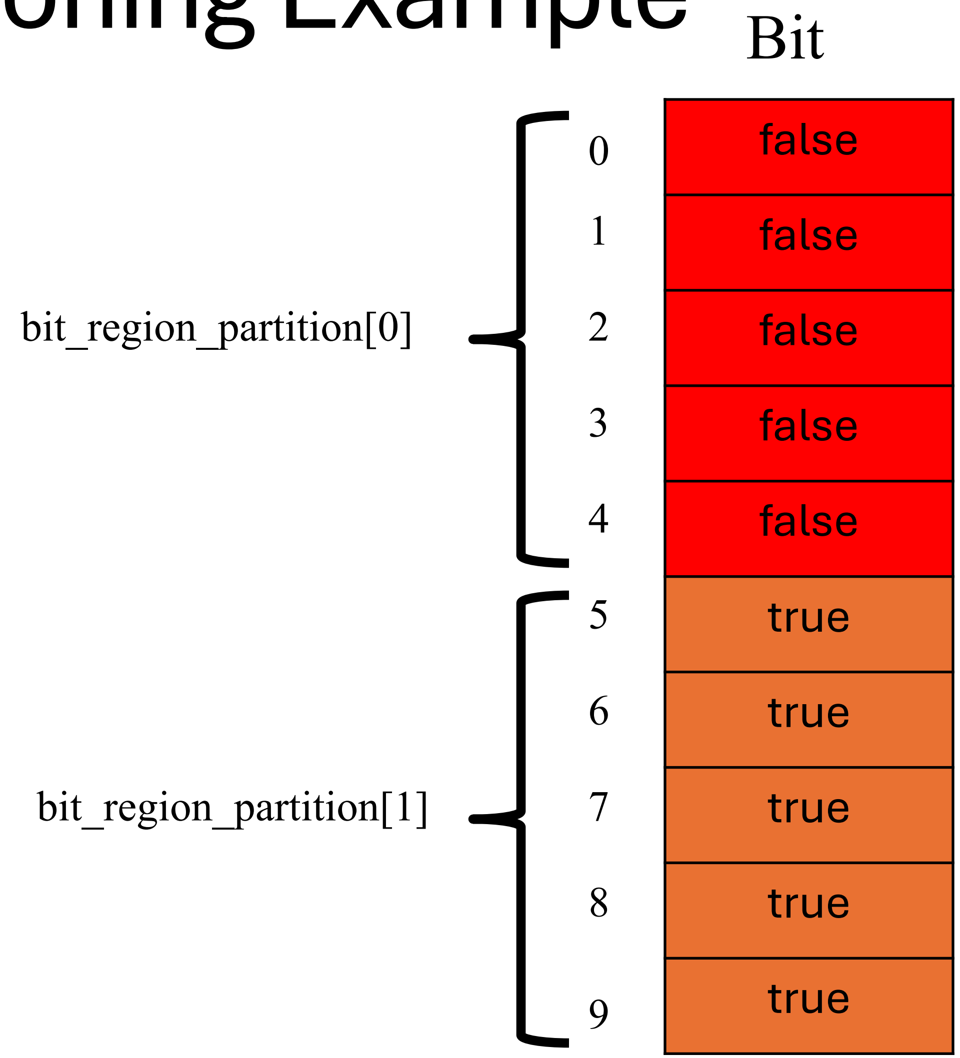
- To enable parallelism on a region, *partition* it into smaller pieces
    - And then run a task on each piece

- Regent has 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	true

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

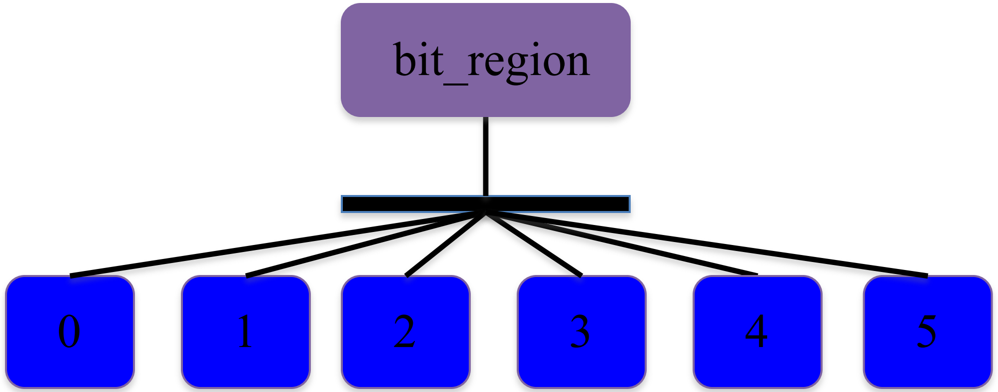
Discussion

- 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

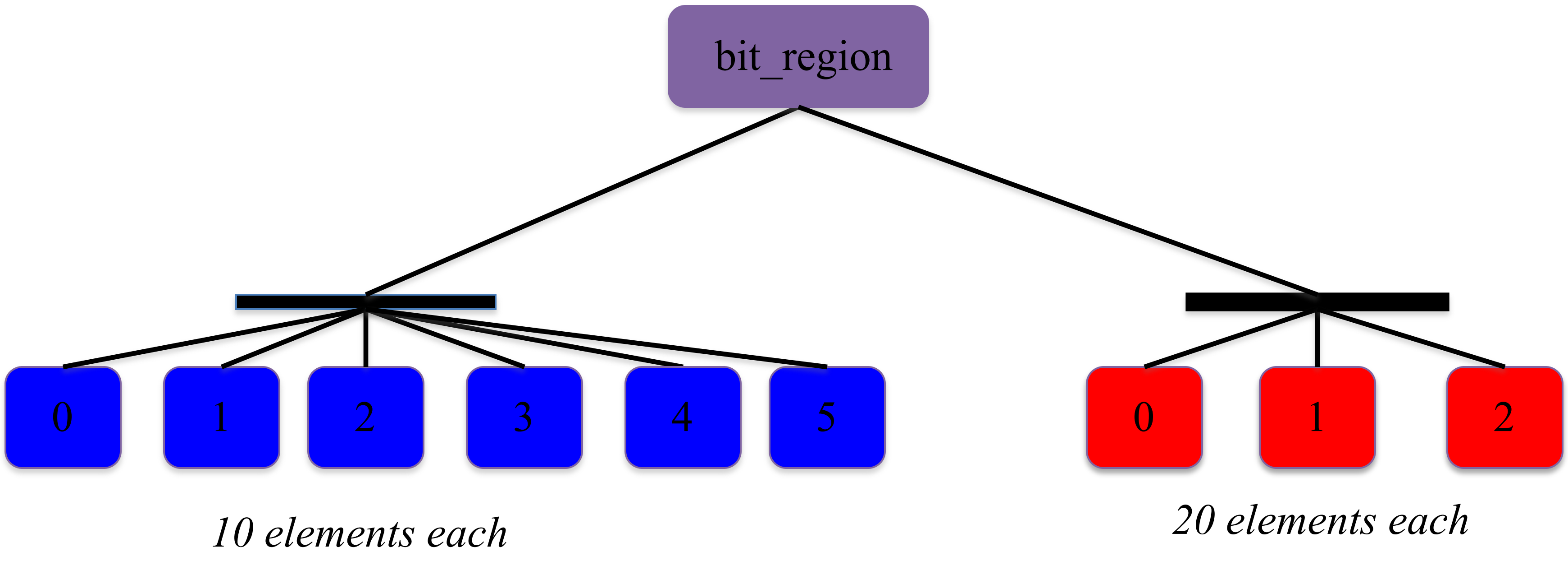
Fills

- 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/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
  - And *coherence,* which we have not discussed

- 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 blur.sh

- Solution is in blur_solution.rg
  - Also scripts for running the solution
  - With and without GPUs

Regions Review

- A region is a (typed) collection

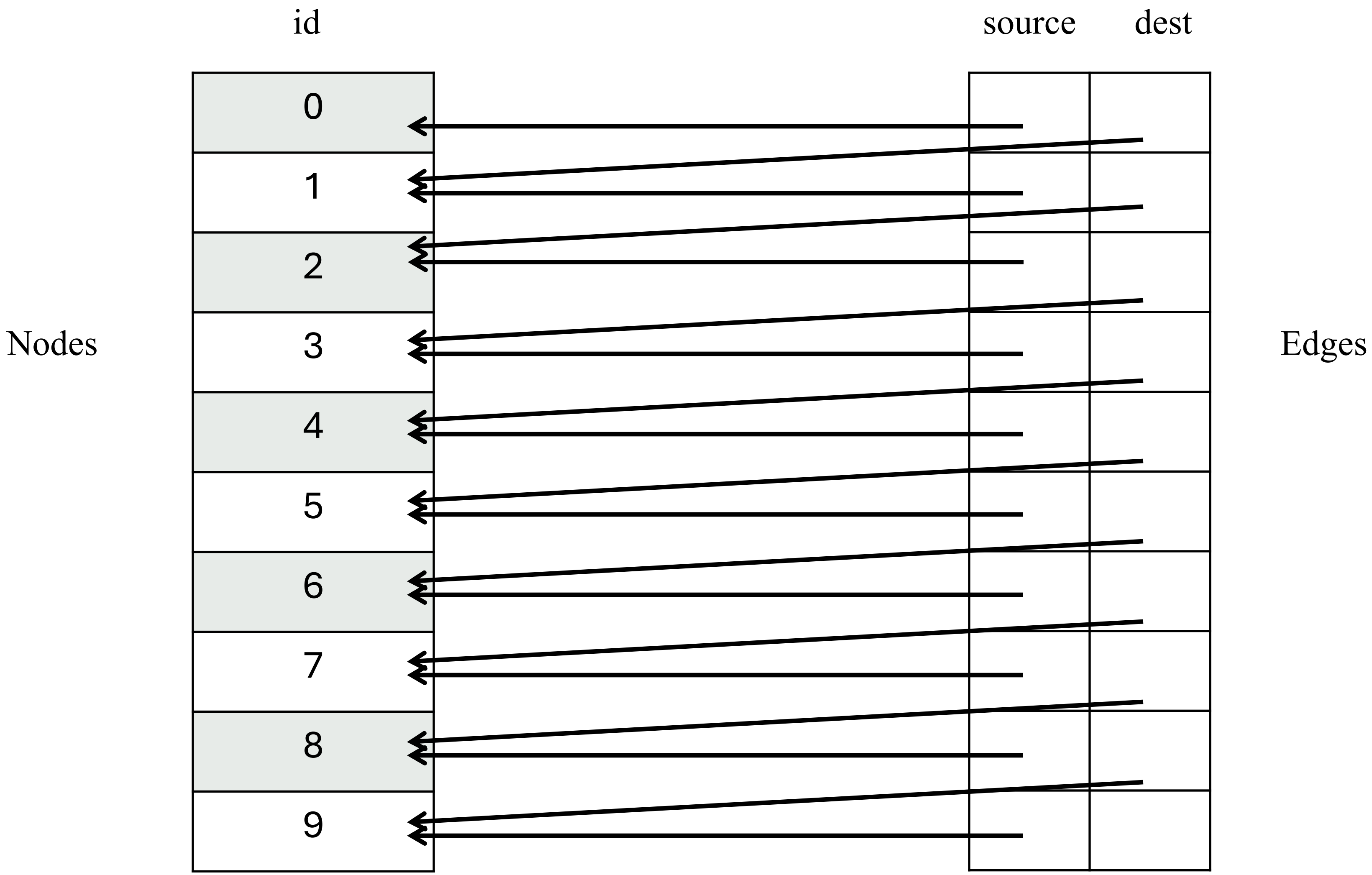
- Regions are the cross product of
  - An *index space*
  - A *field space*

Region References

- Elements of a field can be indices into another region
  - The moral equivalent of a region pointer or reference

- Regent's type system tracks fields that point to other regions
  - E.g., Edge(Nodes) is the type of edges (indices) into a region of Nodes

RegionReferences/1.rg and 2.rg



Partitioning By Field

- A field can be used to create a partition
  - Elements with the same field value are assigned to the same subregion
  - Similar to a "group by" operation in databases

- Write elements of an index space colors into the field f
  - Using an arbitrary computation

- Then call partition(region.f, colors)
  - RegionReferences/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(equal, …)

- *Dependent partitions*
  - Computed using another partition

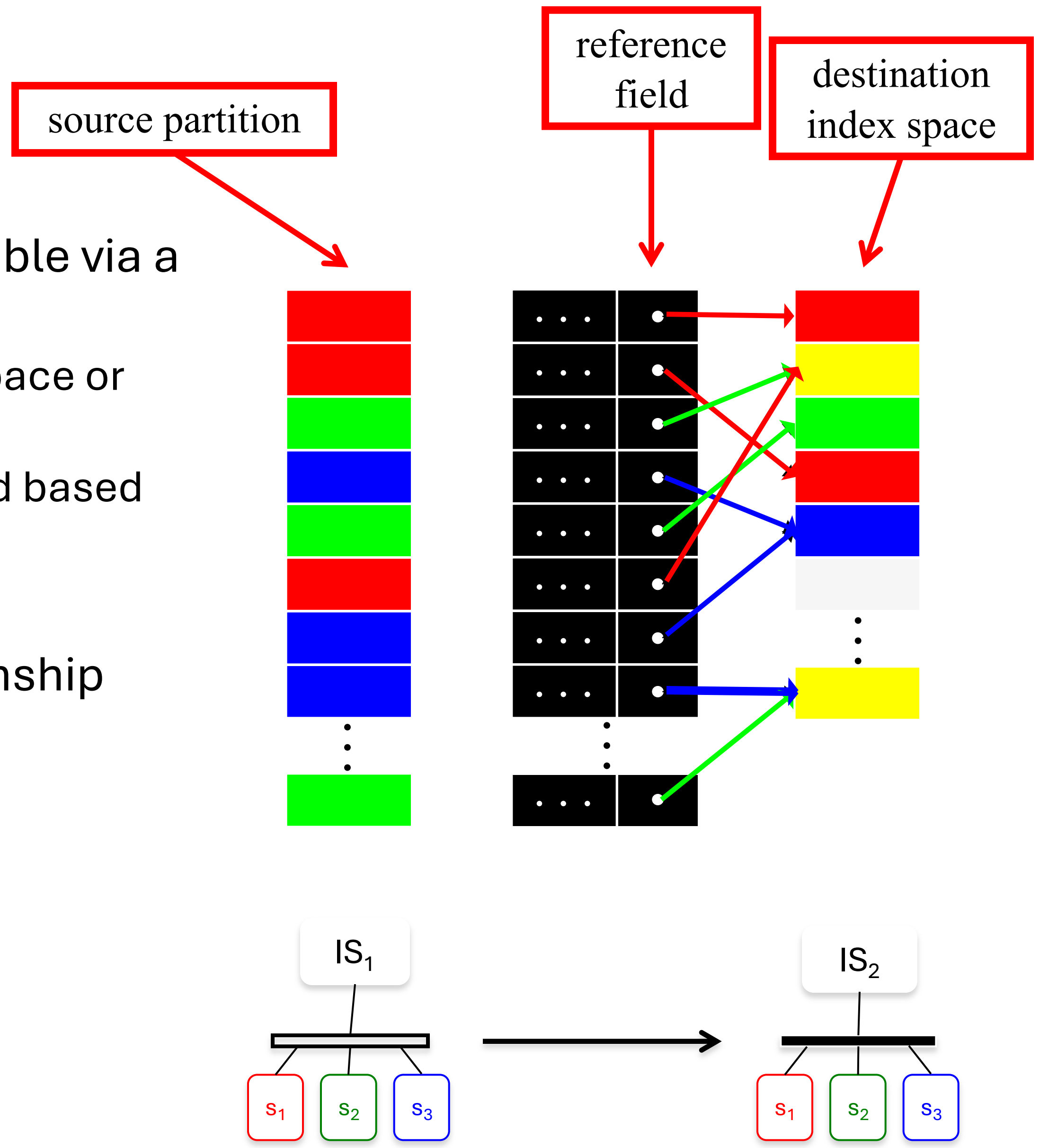
Dependent Partitioning Operations

- 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

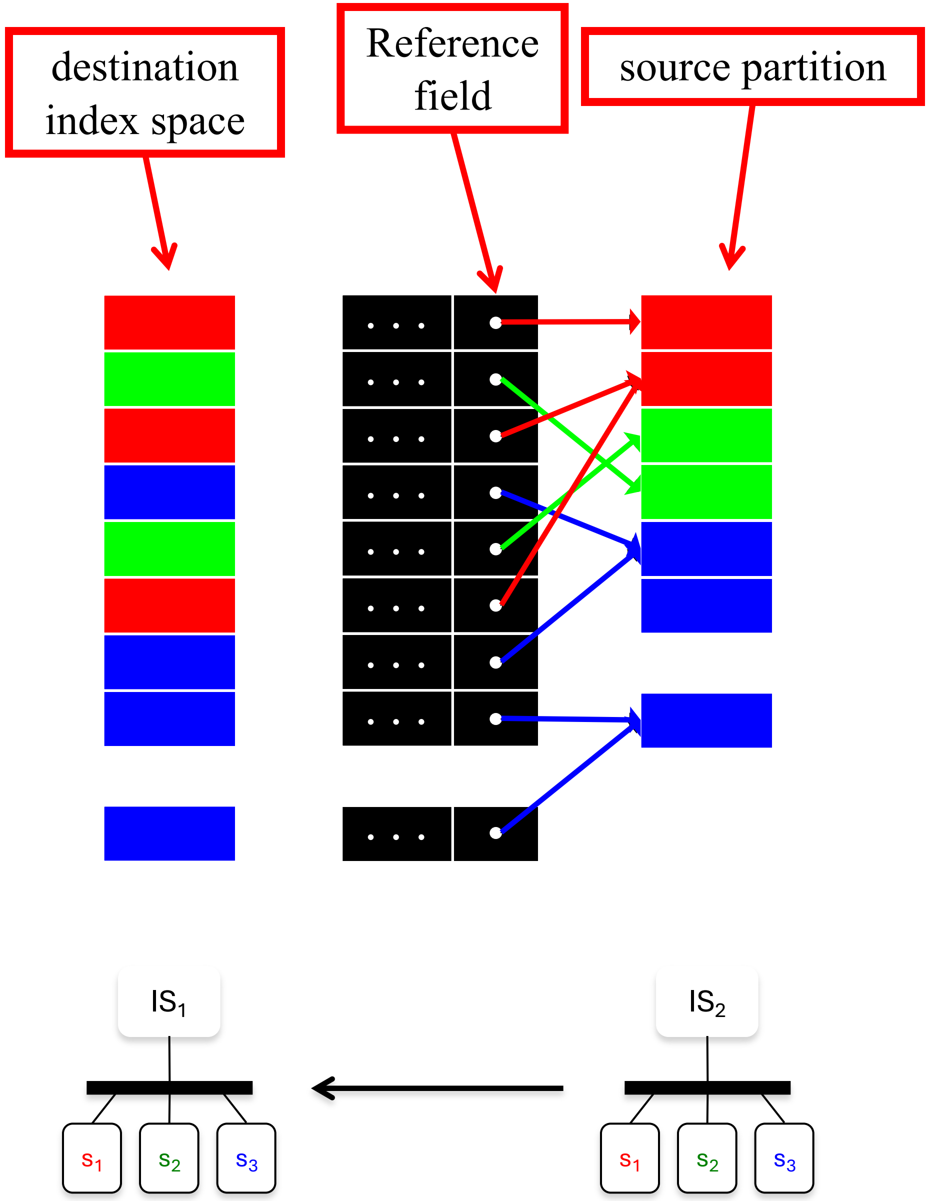
- Regent understands relationship between partitions



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

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
  - Can you switch from a page-based partitioning to a link-based partitioning?
- And possibly the permissions
  - See “TODO”

Control Replication

Subtasks and Scalability

Consider a task

```
task t(...)
    ...
    for color in colors do
        subtask(P[color])
    end
    ...
end
```

If |colors| > ~32, then simply launching the subtasks will be a sequential bottleneck.

Why? The subtasks can run in parallel, but t is sequential.

As |colors| increases, the time to launch the subtasks becomes significant.

Subtasks and Scalability

Consider a task

```
task t(...)
    ...
    for color in colors do
        subtask(P[color])
    end
    ...
end
```

Idea: Spread t's work over multiple processors.

Each launches a subset of the subtasks.

An Issue ...

Consider a task

```
task t(...)
    ...
    i += 1
    var r = region(ispace,fspace)
    format.println(“Hello!”)

    for color in colors do
        subtask(P[color])
    end
end
```

Must preserve t’s semantics!

If t is executed cooperatively by multiple threads, the effect must still be as if t were executed sequentially.

- Index space/region/partition creation happens once
- Print statements happen once

But where possible code should be replicated SPMD-style to avoid communication and synchronization

- Every thread can increment its own local copy of i

Control Replication

To execute a task t cooperatively on multiple processors:

```
__demand(__replicable)
task t(...) ... end
```

The runtime will replicate the task on multiple threads

- With each thread handling a subset of the subtask launches
- And replicating the rest of the task, where it is safe to do so
- ControlReplication/1.rg

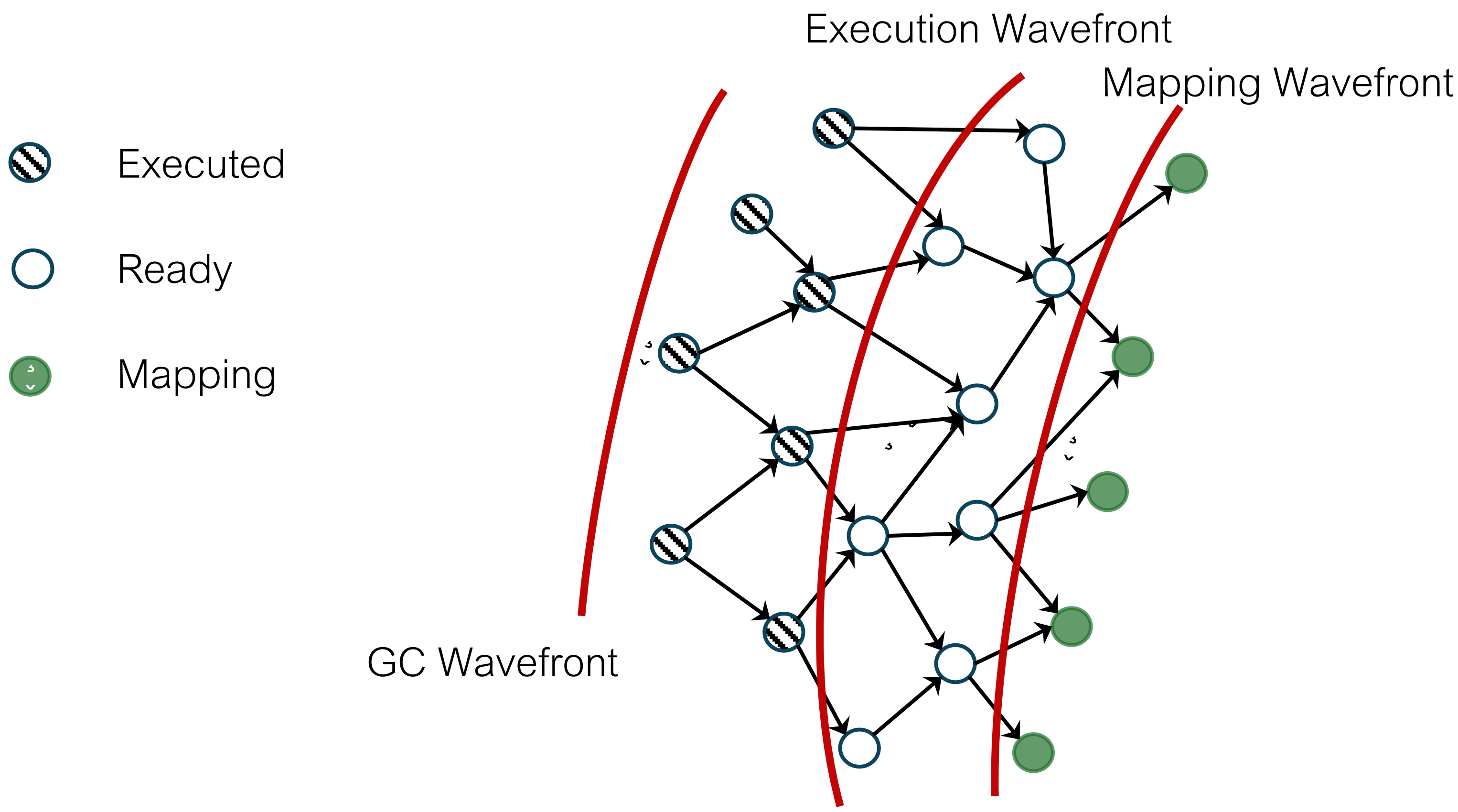
Mapping

Mapping

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

- The runtime issues and maps tasks *ahead of actual execution*
  - In a well-structured program, mapping decisions are made well before it is time to run a task

Picture


Executed
Ready
Mapping
Execution Wavefront
Mapping Wavefront
GC Wavefront

Mapping

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

- Most importantly
  - Assign a processor to each task
    - The task will execute in its entirety on that processor

  - Assign a memory to each region argument

Mappers

- Legion provides a mapping API
  - An interface for making mapping decisions

- The mapping API requires a standard set of functions
  - An implementation of these functions is a *mapper*

- Legion comes with a default mapper
  - A set of heuristics for mapping that makes reasonable decisions
  - But for highest performance applications generally require custom mappers

Task Variants

- A task can have multiple *variants*
  - Different implementations of the same task
  - Multiple variants can be registered with the Legion runtime

- Examples
  - A variant for CPUs
  - Another variant for GPUs
  - Variants for different data layouts

Controlling Processor Mapping in Regent

- By default, the Regent compiler produces only CPU task variants
  - So the default mapper runs these tasks on CPUs

- Place immediately before a task declaration
  - __demand(__cuda)
  - Causes both CPU and GPU task variants to be produced
  - The default mapper prefers a GPU variant if one exists

Where is the Data Placed?

- The default mapper …

- … places region arguments in CPU RAM if the corresponding task is mapped to a CPU
- … places region arguments in the frame buffer if the corresponding task is mapped to a GPU

- There are other possible memories (e.g., zero copy memory) that can be used with a custom mapper

Scratching the Surface

- There are many other mapping options

- Data layout in regions
- Eager/lazy collection of dead regions
- Backpressure – how far ahead of execution should the runtime get?
- …

Libraries

Libraries

- One of the design goals of Legion/Regent is to enable the development of distributed/parallel libraries
  - Composable for both functionality and performance

- Task-based systems are natural for library development
  - Composing independently written tasking programs always makes sense

cuNumeric

- A library from NVIDIA
- Maps NumPy programs to Legion
    - NumPy arrays become regions
    - NumPy array operations are partitioned across the machine

URL: https://developer.nvidia.com/cunumeric

Checkpointing

- A library from SLAC

- Provides checkpointing
  - Low overhead
  - Easy to use
    - in Regent, just add __checkpoint
    - in C++, requires a header and call runtime->checkpoint(...)

https://github.com/StanfordLegion/resilience/

FlexFlow

- A deep learning framework
- Automatically parallelizes deep learning training
  - Searches for a good partitioning of the data
  - Handles model, tensor, data parallelism

- URL: https://flexflow.ai

Research Libraries

- Dense tensor algebra
- Sparse tensor algebra
- Krylov solvers
- FFT
- Mapping DSL

- Happy to work with anyone interested in using these!

Conclusions

Conclusions

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

- Advantages
  - Easy to exploit multiple levels of parallelism in a uniform manner
  - Separate machine mapping
  - Novel and rich partitioning sublanguage
  - Designed to support writing (distributed) libraries

- Good/great performance and portability!