Legion: C++ API Update

Michael Bauer, 03/22/17

NVIDIA

Overview

Introduction and New Features

- Brief C++ Interface Overview
- Legion STL
- New Mapper Interface
- Static Dependences
- Dependent Partitioning
- Dynamic Control Replication

Legion C++ API

Design goals

Why have a C++ API?

- Runtime embedded in an existing (not research) language
- Provide bindings for other languages: C, Lua, Python (coming soon)
- More direct control over what the runtime does

Caveat: C++ here is C++98

Is this still necessary or does everyone have access to C++11/14 compilers?

Legion/Regent Relationship

A simple analogy

Regent Language	High-Level	C Language
Implicit mapping of variables to resources	Implicit calling convention for tasks/functions	More productive
Explicit mapping of variables to resources	Explicit calling convention for tasks/functions	More expressive
Legion Runtime	Low-Level	Assembly Code

Logical and Physical Regions

Names and Resources

In Regent there are just ‘regions’

Legion API distinguishes between ‘logical’ and ‘physical’ regions

Logical regions name collections of data

Physical regions represent a materialization of that data in a memory

Regent manages this relationship for you

In the C++ API it’s your responsibility

```
class LogicalRegion {
public:
  static const LogicalRegion NO_REGION; /**< empty logical region handle*/
protected:
  // Only the runtime should be allowed to make these
  FRIEND_ALL_RUNTIME_CLASSES
  LogicalRegion(RegionTreeID tid, IndexSpace index, FieldSpace field);
public:
  LogicalRegion(void);
  LogicalRegion(const LogicalRegion &rhs);
public:
  inline LogicalRegion& operator=(const LogicalRegion &rhs);
```

```
class PhysicalRegion {
public:
  PhysicalRegion(void);
  PhysicalRegion(const PhysicalRegion &rhs);
  ~PhysicalRegion(void);
private:
  Internal::PhysicalRegionImpl *impl;
protected:
  FRIEND_ALL_RUNTIME_CLASSES
  explicit PhysicalRegion(Internal::PhysicalRegionImpl *impl);
public:
  PhysicalRegion& operator=(const PhysicalRegion &rhs);
```

Legion Tasks

A generic interface for all computations

All Legion tasks have the same type

User responsible for packing/unpacking arguments into this format

Data structure that contains task meta-data

Mapped physical regions requested for the execution of this task (order is user defined)

```
void hello_world_task(const Task *task,
                      const std::vector<PhysicalRegion> &regions,
                      Context ctx, Runtime *runtime)
```

Regent compiler packs and unpacks all arguments for you

Opaque handle used for launching sub-tasks

Pointer to the Legion runtime

Launching Tasks

Launchers and Region Requirements

All operations created with launcher structures

```
struct TaskLauncher {
public:
  TaskLauncher(void);
  TaskLauncher(Processor::TaskFuncID tid,
               TaskArgument arg,
               Predicate pred = Predicate::TRUE_PRED,
               MapperID id = 0,
               MappingTagID tag = 0);
public:
  inline IndexSpaceRequirement&
        add_index_requirement(const IndexSpaceRequirement &req);
  inline RegionRequirement&
        add_region_requirement(const RegionRequirement &req);
  inline void add_field(unsigned idx, FieldID fid, bool inst = true);
public:
  inline void add_future(Future f);
```

Region requirements specify logical regions and privileges requested

```
struct RegionRequirement {
public:
  RegionRequirement(void);
  /**
   * Standard region requirement constructor for logical region
   */
  RegionRequirement(LogicalRegion _handle,
                    const std::set<FieldID> &privilege_fields,
                    const std::vector<FieldID> &instance_fields,
                    PrivilegeMode _priv, CoherenceProperty _prop,
                    LogicalRegion _parent, MappingTagID _tag = 0,
                    bool _verified = false);
```

Accessors and Raw Pointers

Getting access to data in physical regions

Two ways to get access to data in physical regions

- Accessors
- Raw pointers

Can be verbose

Accessors have some overhead but provide safety checks

Raw pointers are fast but unsafe

```
RegionAccessor<AccessorType::Generic, double> acc =
  regions[0].get_field_accessor(fid).typeify<double>();

Domain dom = runtime->get_index_space_domain(ctx,
    task->regions[0].region.get_index_space());
Rect<1> rect = dom.get_rect<1>();
for (GenericPointInRectIterator<1> pir(rect); pir; pir++)
{
  acc.write(DomainPoint::from_point<1>(pir.p), drand48());
}
```

```
Rect<2> subrect;
LegionRuntime::Accessor::ByteOffset offsets[2];
void *data = handle->raw_rect_ptr<2>(rect, subrect, &offsets[0]);
```

Legion STL

Library of common Legion template patterns

Started a collection of common template patterns that Legion users employ

Task wrappers for unpacking raw pointers for each field of a physical region

(Up to 16 regions)

Open to suggestions

C++11/14 supported

Vector of pointers for each region requirement with 1 entry per field

Byte offsets for each physical region

```
template<typename T0, int DIM0, typename T1, int DIM1,
  void (*PTR)(const Task*, Context, Runtime*,
              const std::vector<T0*>&, const ByteOffset[DIM0],
              const std::vector<T1*>&, const ByteOffset[DIM1])>
static void raw_rect_task_wrapper(const Task *task,
   const std::vector<PhysicalRegion>& regions, Context ctx, Runtime *runtime);
```

New Mapping Interface

As promised at last year's bootcamp

New mapping interface is now live

Mapper calls all have the same format

Easier to tell inputs and outputs

Explicit management of physical instances

Set constraints for describing layouts

```
bool create_physical_instance(
                              MapperContext ctx, Memory target_memory,
                              const LayoutConstraintSet &constraints,
                              const std::vector<LogicalRegion> &regions,
                              PhysicalInstance &result, bool acquire=true,
                              GCPriority priority = 0) const;
```

```
struct MapTaskInput {
  std::vector<std::vector<PhysicalInstance> >     valid_instances;
  std::vector<unsigned>                           premapped_regions;
};
struct MapTaskOutput {
  std::vector<std::vector<PhysicalInstance> >     chosen_instances;
  std::vector<Processor>                          target_procs;
  VariantID                                       chosen_variant; // = 0
  ProfilingRequest                                task_prof_requests;
  ProfilingRequest                                copy_prof_requests;
  TaskPriority                                    task_priority;  // = 0
  bool                                            postmap_task; // = false
};
//------------------------------------------------------------------------
virtual void map_task(const MapperContext         ctx,
                      const Task&                 task,
                      const MapTaskInput&         input,
                            MapTaskOutput&        output) = 0;
//------------------------------------------------------------------------
```

Context for runtime calls | Task Meta-data | Input Struct | Output Struct

New Default Mapper Implementation

Making it easier to influence policy

New default mapper implementation for new mapper interface

Some better heuristics and policies

Mapper is more complex so look for ‘default_policy_’ methods to overload

Easy to create custom mappers while using default machinery

```
virtual Processor default_policy_select_initial_processor(
                                MapperContext ctx, const Task &task);
virtual void default_policy_select_target_processors(
                                MapperContext ctx,
                                const Task &task,
                                std::vector<Processor> &target_procs);
virtual bool default_policy_select_must_epoch_processors(
                                MapperContext ctx,
                                const std::vector<std::set<const Task *> > &tasks,
                                Processor::Kind proc_kind,
                                std::map<const Task *, Processor> &target_procs);
virtual void default_policy_rank_processor_kinds(
                                MapperContext ctx, const Task &task,
                                std::vector<Processor::Kind> &ranking);
virtual VariantID default_policy_select_best_variant(MapperContext ctx,
                                const Task &task, Processor::Kind kind,
                                VariantID vid1, VariantID vid2,
                                const ExecutionConstraintSet &execution1,
                                const ExecutionConstraintSet &execution2,
                                const TaskLayoutConstraintSet &layout1,
                                const TaskLayoutConstraintSet &layout2);
virtual Memory default_policy_select_target_memory(MapperContext ctx,
                                Processor target_proc);
```

Static Dependences

Communicating static information

```
void begin_static_trace(Context ctx,
                        const std::set<RegionTreeID> *managed = NULL);
```

Provide interface to communicate statically known dependence information

Reduce runtime overhead

Wrap code blocks in begin/end_static_trace

Describe static operations for each task

Pass pointer to dependences on launchers

```
public:
  // Inform the runtime about any static dependences
  // These will be ignored outside of static traces
  const std::vector<StaticDependence> *static_dependences;
```

```
struct StaticDependence {
public:
  StaticDependence(void);
  StaticDependence(unsigned previous_offset,
                   unsigned previous_req_index,
                   unsigned current_req_index,
                   DependenceType dtype,
                   bool validates = false);
public:
  inline void add_field(FieldID fid);
public:
  // The relative offset from this operation to
  // previous operation in the stream of operations
  // (e.g. 1 is the operation launched immediately before)
  unsigned                previous_offset;
  // Region requirement of the previous operation for the dependence
  unsigned                previous_req_index;
  // Region requirement of the current operation for the dependence
  unsigned                current_req_index;
  // The type of the dependence
  DependenceType          dependence_type;
  // Whether this requirement validates the previous writer
  bool                    validates;
  // Fields that have the dependence
  std::set<FieldID>       dependent_fields;
};
```

Dependent Partitioning API

Better ways to compute partitions

Development branch 'deppart'

Will merge to master in 3-4 weeks

Almost fully backwards compatible

Partitions no longer computed with colorings

Create partitions from field data…

… or based on other partitions

Deferred computations just like all other Legion operations

```
IndexPartition create_partition_by_field(Context ctx,
                                         LogicalRegion handle,
                                         LogicalRegion parent,
                                         FieldID fid,
                                         IndexSpace color_space,
                                         Color color = AUTO_GENERATE_ID,
                                         MapperID id = 0,
                                         MappingTagID tag = 0);
```

```
Color create_cross_product_partitions(Context ctx,
                                      IndexPartition handle1,
                                      IndexPartition handle2,
                                      std::map<IndexSpace,IndexPartition> &handles,
                                      PartitionKind part_kind = COMPUTE_KIND,
                                      Color color = AUTO_GENERATE_ID);
```

Dependent Partitioning (Part 2)

Templated Index Spaces and Logical Regions

New support for templated index spaces, partitions, and logical regions

- Integer dimension
- Coordinate type

Inherit from non-templated base type

Templated versions of runtime calls

```
template<int DIM, typename COORD_T,
         int COLOR_DIM, typename COLOR_COORD_T>
IndexPartitionT<DIM,COORD_T> create_partition_by_field(Context ctx,
                    LogicalRegionT<DIM,COORD_T> handle,
                    LogicalRegionT<DIM,COORD_T> parent,
                    FieldID fid, // type: ZPoint<COLOR_DIM,COLOR_COORD_T>
                    IndexSpaceT<COLOR_DIM,COLOR_COORD_T> color_space,
                    Color color = AUTO_GENERATE_ID,
                    MapperID id = 0, MappingTagID tag = 0);
```

```
template<int DIM, typename COORD_T>
class IndexSpaceT : public IndexSpace {
protected:
  // Only the runtime should be allowed to make these
  FRIEND_ALL_RUNTIME_CLASSES
  IndexSpaceT(IndexSpaceID id, IndexTreeID tid);
public:
  IndexSpaceT(void);
  IndexSpaceT(const IndexSpaceT &rhs);
  explicit IndexSpaceT(const IndexSpace &rhs);
};
```

```
template<int DIM, typename COORD_T>
class LogicalRegionT : public LogicalRegion {
protected:
  // Only the runtime should be allowed to make these
  FRIEND_ALL_RUNTIME_CLASSES
  LogicalRegionT(RegionTreeID tid, IndexSpace index, FieldSpace field);
public:
  LogicalRegionT(void);
  LogicalRegionT(const LogicalRegionT &rhs);
  explicit LogicalRegionT(const LogicalRegion &rhs);
};
```

A Revisionist History of Legion S3D

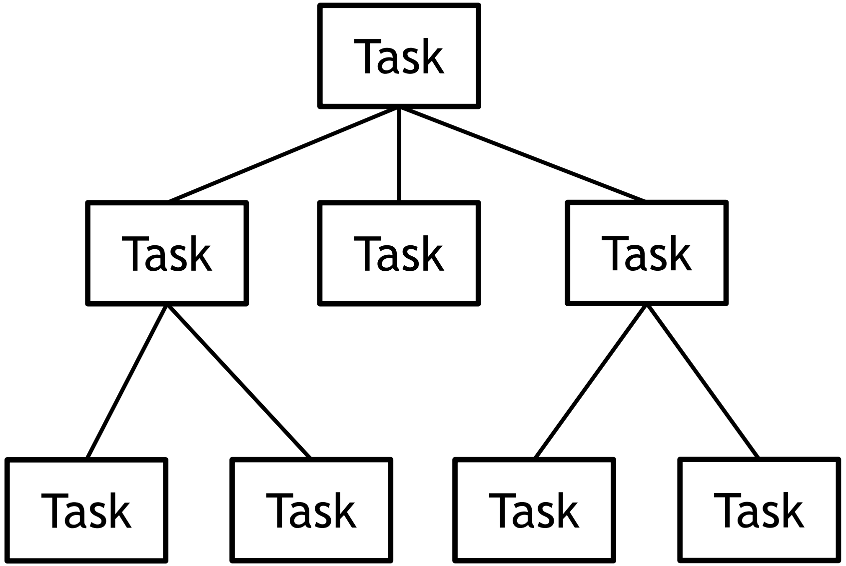
The two versions

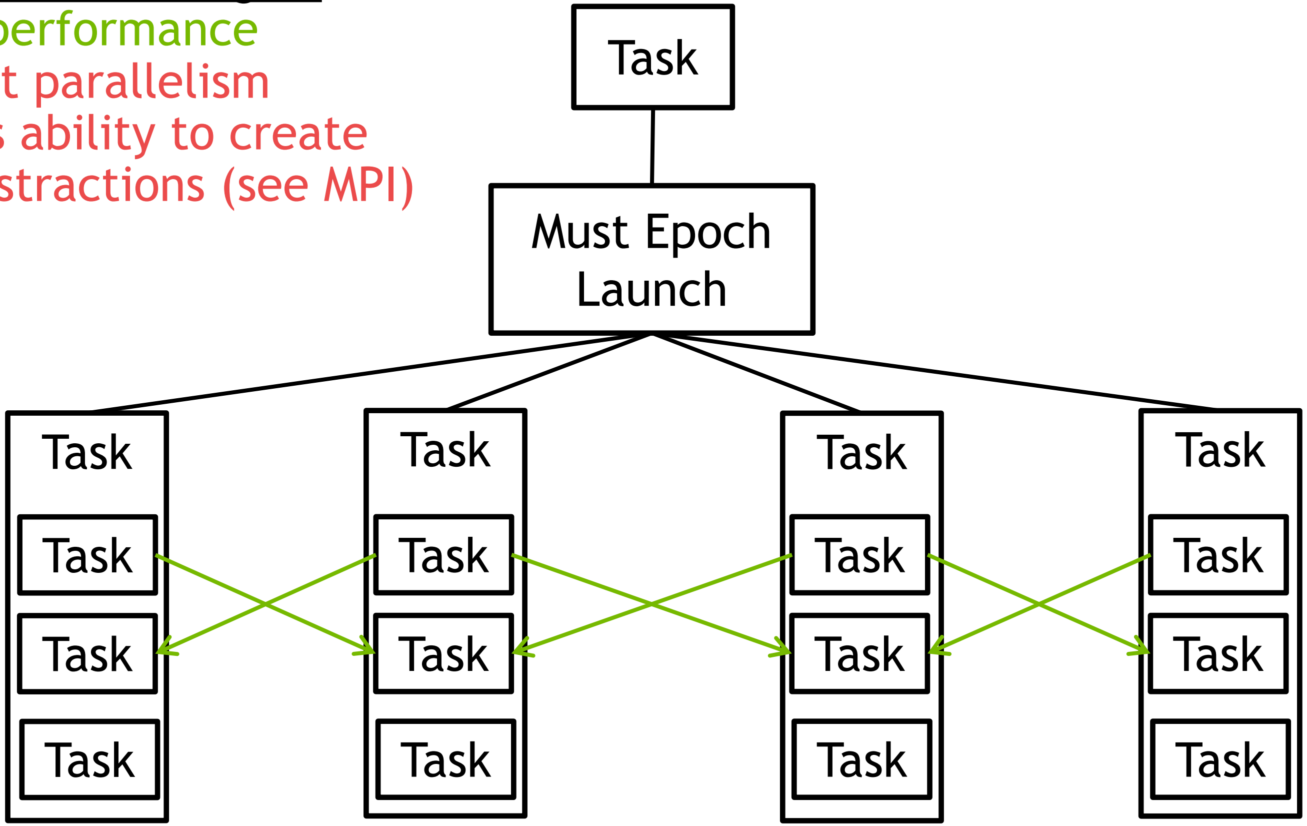
Pure Legion

- Good programmability
- Didn't have experience necessary to build it and make it scale

Extended Legion

- Good performance
- Explicit parallelism destroys ability to create good abstractions (see MPI)





The Problem

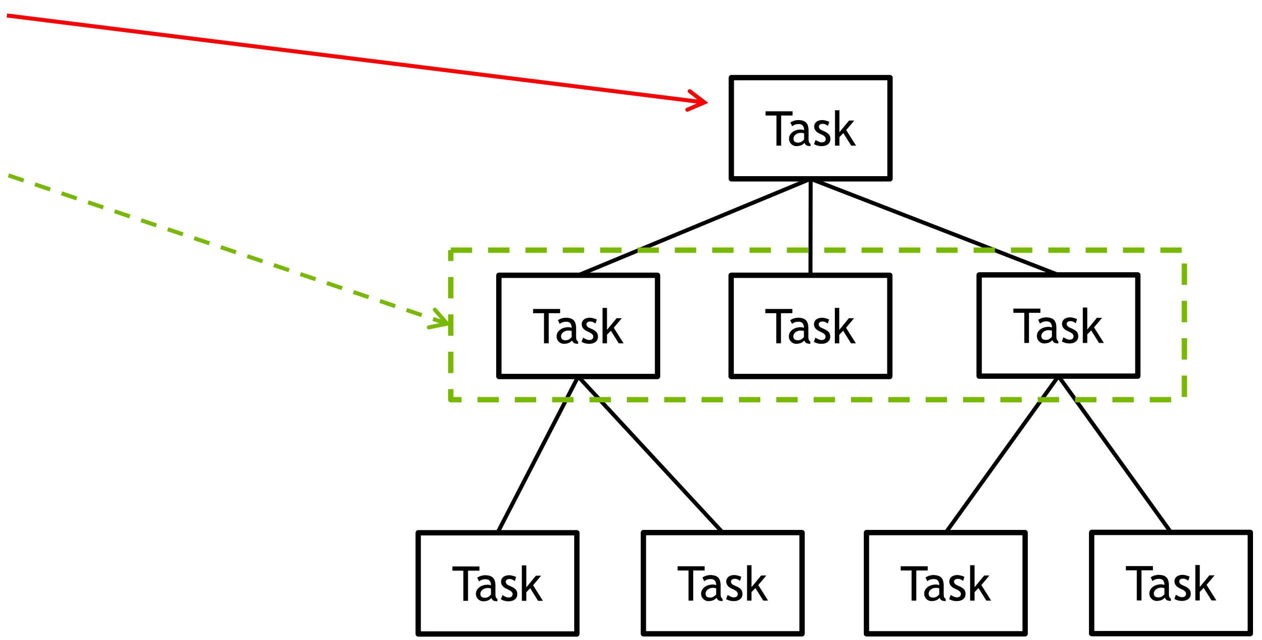
How do we make this scale?

This task can only run on one node

What if it has to launch many subtasks per iteration?

Fact: no matter how efficient the program analysis is, at some granularity of task and number of nodes it will become a sequential bottleneck

True for “all” interesting Legion applications at “scale”



“Short Term” Hack

Must Epoch Launchers and Phase Barriers

Temporary solution: must epoch task launch

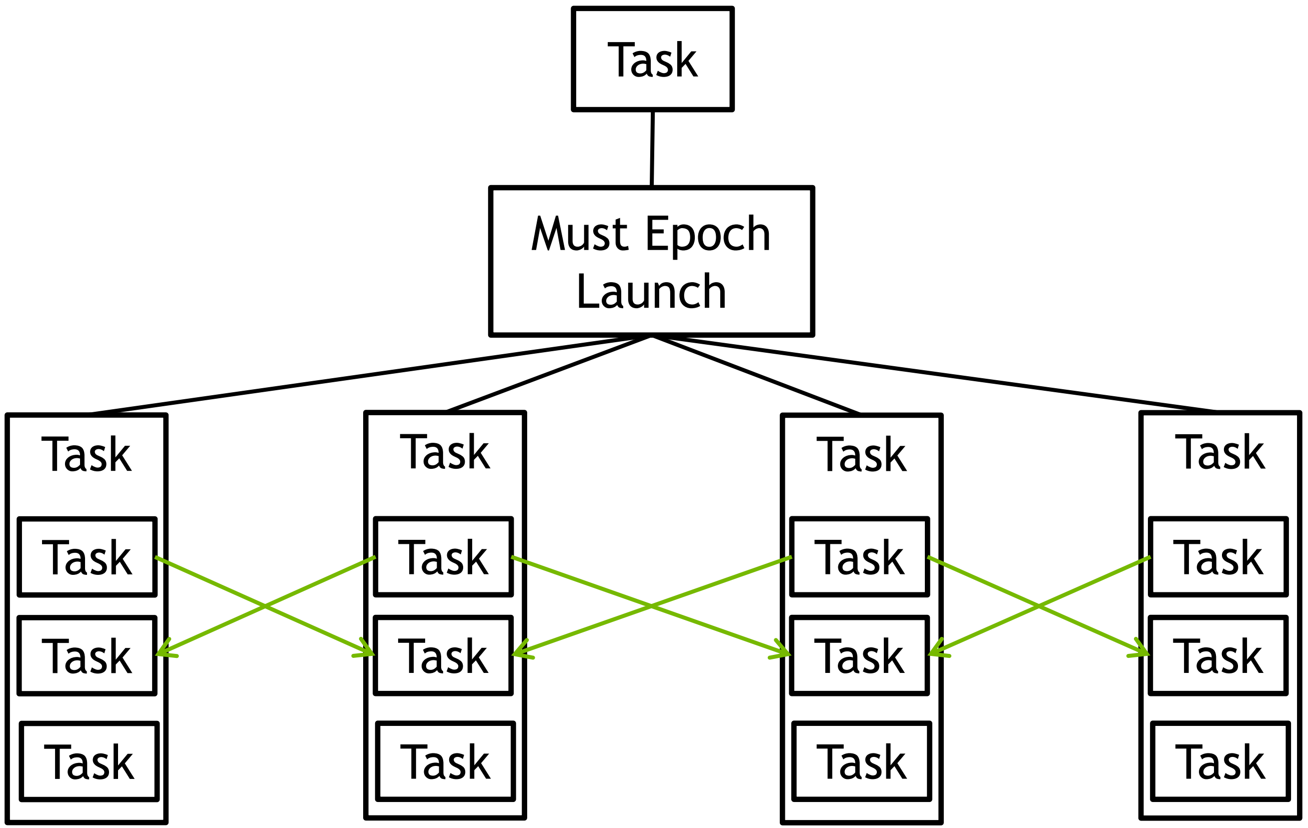
Long running tasks communicate through shard regions

Synchronize with phase barriers

Problem 1: fixed communication patterns only

Problem 2: must epoch still has sequential launch overhead

Not very Legion-like ☹



Why is this a hack?

Software Composability

Today: MPI / Must-Epoch Style

```
mpirun / must epoch {
    task {
        while (true) {
          for (all whatever)
             compute phase1
          explicit communication/sync
          for (all whatever)
             compute phase 2
          explicit communication/sync
          …
        }
    }
}
```

Ideal Sequential Code

```
while (true) {
    for (all whatever)
        compute phase 1
    for (all whatever)
        compute phase 2
    …
}
```

Legion (w/ Control Replication)

```
task {
    while (true) {
        Index task launch phase 1
        Index task launch phase 2
        …
    }
}
```

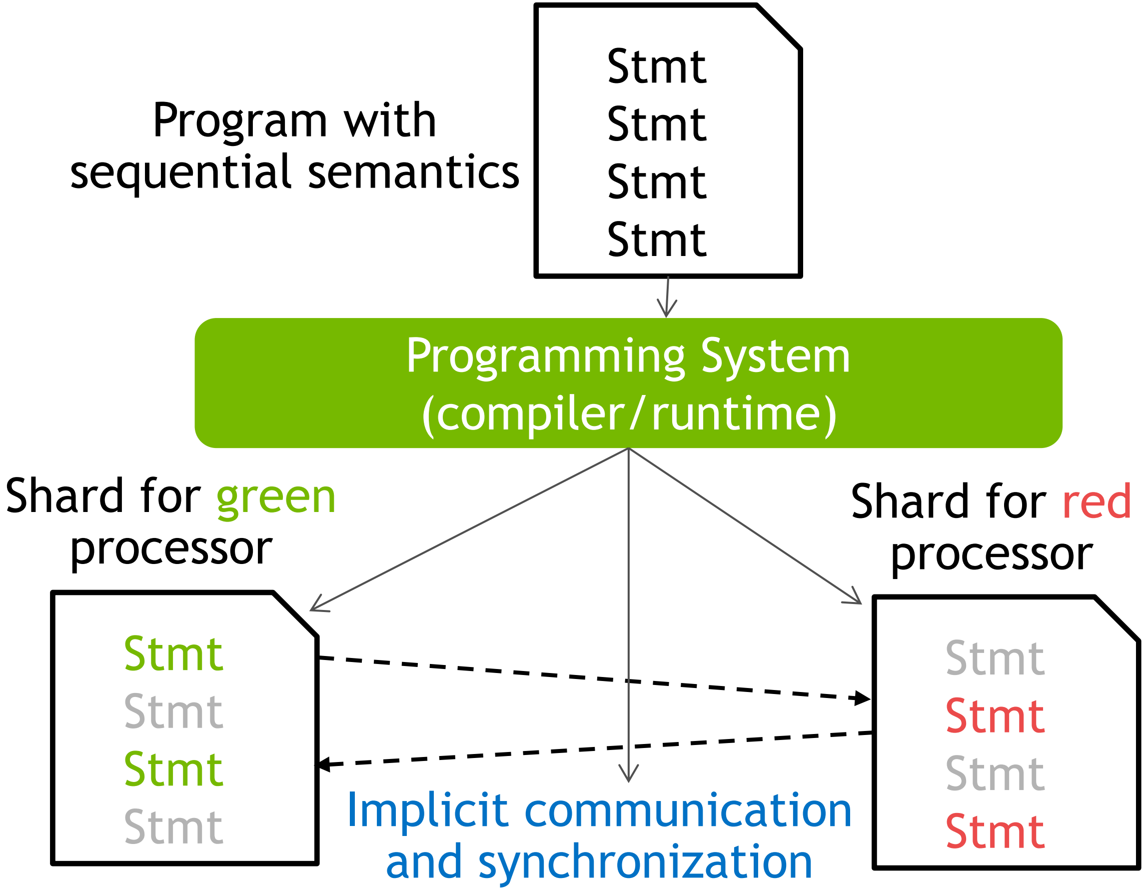
Nasty explicit communication and synchronization

No explicit communication or synchronization

Can we make this scale?

Control Replication

Scalable Implicit Parallelism



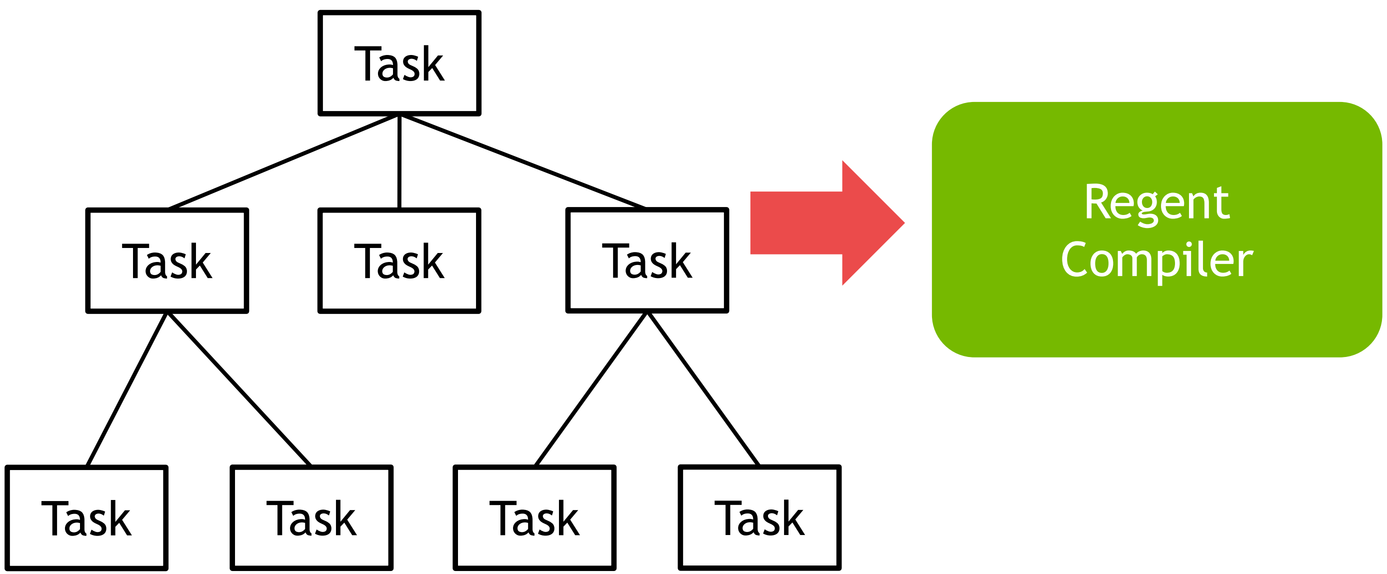
Two variations on this:

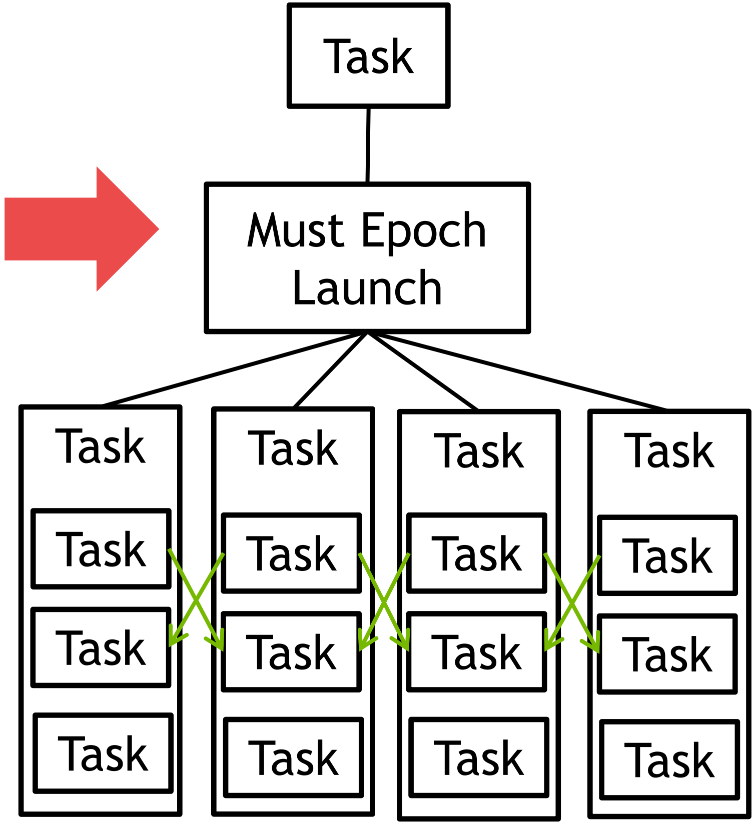
Static Control Replication (Regent)

Dynamic Control Replication (Legion)

Static Control Replication

Implementation in Regent





Static Analysis

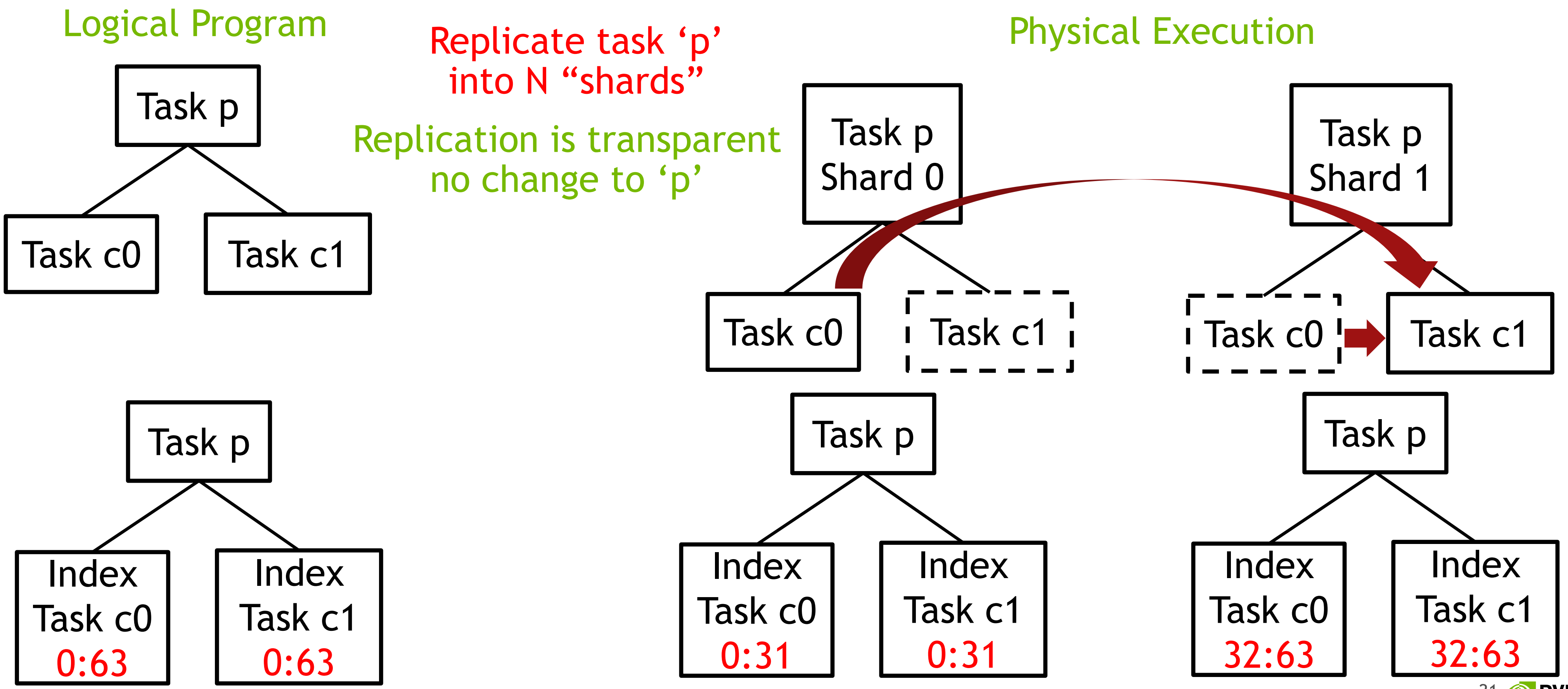
Pro: zero overhead, good performance

Con: can only handle “partially” static communication

Insufficient for things like AMR and AMG

Dynamic Control Replication

Handling dynamic program behavior



Replicable Task Variants

Task Variant Requirements

Legion task variants have properties (e.g. leaf, inner, idempotent)

We will add a 'replicable' property

No side effects (e.g. call random number generator, maybe no printf statements)

All operations must be annotated with two fields to map to shards:

- Point (single ops) or Domain (index ops)
- Slicing functor (more on next slide)

```
struct TaskLauncher {
   …
   DomainPoint          index_point;
   ShearingID           shearing_functor;
   …
};
```

```
struct IndexLauncher {
   …
   Domain               index_domain;
   ShearingID           shearing_functor;
   …
};
```

Slicing Functors

Determining which shards own which operations

Create slicing functors just like current projection functors

Runtime will invoke functor on each operation launched in replicated task

Can define arbitrary cleaving functions

Must be "functional"

Design questions: what kinds of methods must a slicing functor support?

```
class SlicingFunctor {

  // We definitely want this one
  virtual ShardID slice(Point p) = 0;

  // Can we do the inverse too?
  virtual void inverse_slice(ShardID id,
     Domain d, set<Point> &points) = 0;

  virtual bool is_exclusive(void) const = 0;
};
```

Reminder: slicing functions just say which shard owns an operation, not where it maps

New Operation Kinds

Index Launches for Everything

Single Operation Kinds:	Index Space Operation Kinds:	
Task	Index Task	Use normal projection functions
Fill	Index Fill	
(Dependent) Partition	(More on partitioning soon)	
Region-to-Region Copy	Index Region-to-Region Copy	Will do these operations on demand
Acquire/Release	Index Acquire/Release	
Attach/Detach	Index Attach/Detach	
Inline Mapping	Nope! (why not?)	

"Collectives"

Existing Legion features provide collective-like behavior

Logical Program

```
FutureMap fm = index_space_launch(…);

// Launch sub operations dependent on futures
```

All-to-all functionality

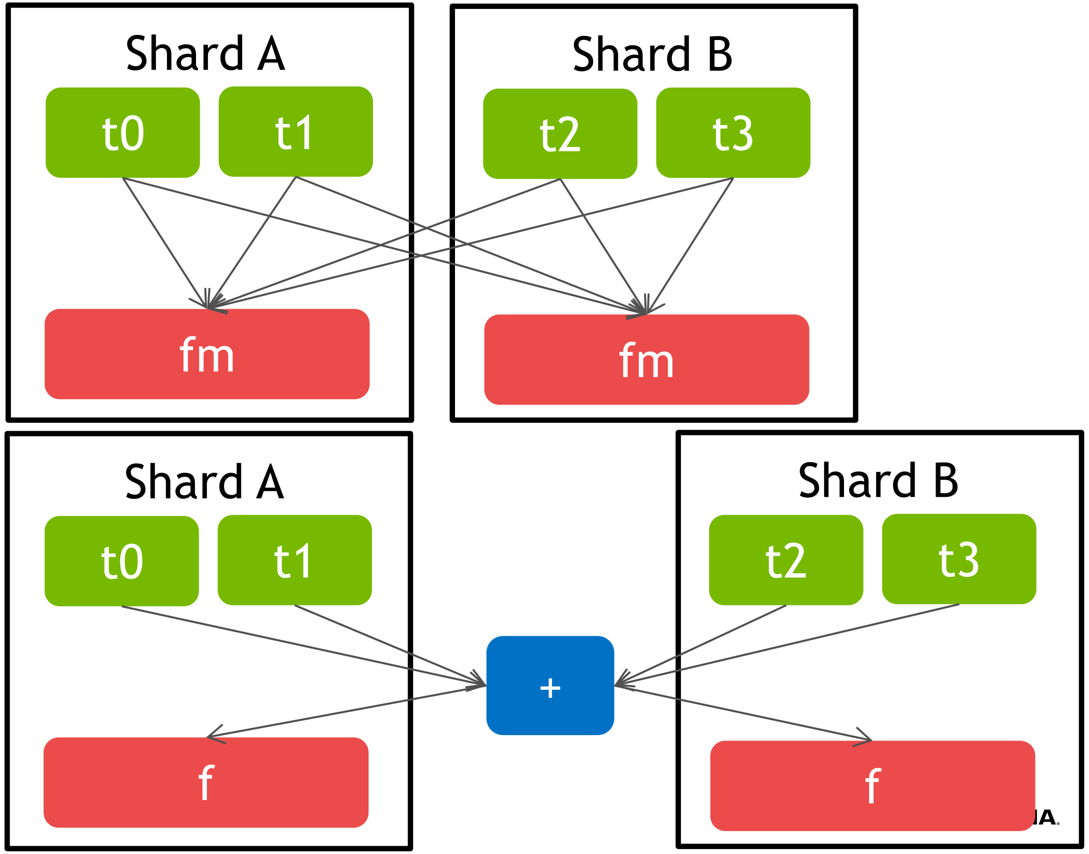
... only better because we can do it lazily

```
Future f = index_space_launch(…, reduction:+);
```

All-reduce functionality

... can be lazy here too

Physical Execution



Creating Regions and Partitions

Making sure things are symmetric

Other runtime operations must be implemented as “collectives”

Each shard must get the same name

What about (dependent) partitioning?

Must also be internal “collective”

Still debating the best way to implement this between Legion and Realm

- Alternative 1: partial partitioning
- Alternative 2: reduce to one shard

```
IndexSpace is = create_index_space(…)

FieldSpace fs = create_field_space(…)

LogicalRegion lr = create_logical_region(..)
```

```
IndexPartition ip = create_equal_partition(…)

IndexPartition ip = create_weighted_partition(…)

IndexPartition ip = create_partition_by_field(..)

IndexPartition ip = create_partition_by_image()

IndexPartition ip = create_partition_by_preimage()
```

Mapper Extensions

Only one mapper call to change

Modify map_task mapper call output

Chosen variant can be replicable

Will ignore 'num_shards' if not replicable

Shards assigned to processors in vector

Initially will only support control replication for top-level task

```
struct MapTaskOutput {
  vector<vector<PhysicalInstance>> instances;
  vector<Processor>                processors;
  VariantID                        variant;
  ProfilingRequestSet              requests;
  TaskPriority                     priority;
  bool                             postmap;
  unsigned                         num_shards;
};
```

Implementation Details

Planned Phases

Step 1: Refactor close operations to make them efficient (done!)

Step 2: Make ‘control_replication’ branch (done!)

Step 3: Update interface for development (done!)

Step 4: Data-parallel-only control replication (in progress)

- Replicate tasks, index launches, replication functions, no communication

Step 5: Introduce communication (in progress)

- Make close operations work

Step 6: Add support for additional index launch operations as needed

The Vision

Scalable and Composable Software with Sequential Semantics

```
task top_level {
    call into legion_metis
    for (however long) {
        call into legion_boxlib
        call into legion_hyper
        call into legion_...
    }
}
```

IS task launch
dependent partition
...

IS task launch
IS task launch
...
IS task launch
IS task launch
...
IS task launch
IS task launch
...

No explicit communication

No explicit synchronization

Scale to 10K+ nodes

NVIDIA