STATE OF

Legion

Michael Bauer, 12/07/22

10 YEARS OF LEGION

Past the great filter

Thank you to everyone who helped us get here!

"All systems software needs 10 years to mature before it is robust." -- Alex

We're pretty much right on time…

Legion: Expressing Locality and Independence with Logical Regions

Michael Bauer, Stanford University, mebauer@cs.stanford.edu
Sean Treichler, Stanford University, sjt@cs.stanford.edu
Elliott Slaughter, Stanford University, slaughter@cs.stanford.edu
Alex Aiken, Stanford University, aiken@cs.stanford.edu

***Abstract*—Modern parallel architectures have both heterogeneous processors and deep, complex memory hierarchies. We present Legion, a programming model and runtime system for achieving high performance on these machines. Legion is organized around *logical regions*, which express both locality and independence of program data, and *tasks*, functions that perform computations on regions. We describe a runtime system that dynamically extracts parallelism from Legion programs, using a distributed, parallel scheduling algorithm that identifies both independent tasks and nested parallelism. Legion also enables explicit, programmer controlled movement of data through the memory hierarchy and placement of tasks based on locality information via a novel mapping interface. We evaluate our Legion implementation on three applications: fluid-flow on a regular grid, a three-level AMR code solving a heat diffusion equation, and a circuit simulation.**

I. Introduction

Modern parallel machines are increasingly complex, with deep, distributed memory hierarchies and heterogeneous processing units. Because the costs of communication within these architectures vary by several orders of magnitude, the penalty for mistakes in the placement of data or computation is usually very poor performance. Thus, to achieve good performance the programmer and the programming system must reason about *locality* (data resident close to computation that uses it) and *independence* (computations operating on disjoint data, and therefore not requiring communication and able to be placed in possibly distant parts of the machine). Most contemporary programming systems have no facilities for the programmer to express locality and independence. The few languages that do focus primarily on array-based locality [1], [2], [3] and avoid irregular pointer data structures.

In this paper we describe Legion, a parallel programming system based on using *logical regions* to describe the organization of data and to make explicit relationships useful for reasoning about locality and independence. A logical region names a set of objects. Logical regions are first-class values in Legion and may be dynamically allocated, deleted and stored in data structures. Regions can also be passed as arguments to distinguished functions called *tasks* that access the data in those regions, providing locality information. Logical regions may be *partitioned* into disjoint or aliased (overlapping) *sub-regions*, providing information for determining independence of computations. Furthermore, computations access logical regions with particular *privileges* (*read-only*, *read-write*, and *reduce*) and *coherence* (e.g., *exclusive access* and *atomic access*, among others). Privileges express how a task may use its region arguments, providing data dependence information that is used to guide the extraction of parallelism. For example, if two tasks access the same region with read-only privileges the two tasks can potentially be run in parallel. Coherence properties express the required semantics of concurrent region accesses. For example, if the program executes $f_1(r); f_2(r)$ and tasks f_1 and f_2 both require *exclusive* access to region r, then Legion guarantees the result will be as if $f_1(r)$ completes before $f_2(r)$ begins. On the other hand, if the tasks access r with *atomic* coherence, then Legion guarantees only atomicity of the tasks with respect to r: either task $f_1(r)$ appears to run entirely before $f_2(r)$ or vice versa.

Logical regions do not commit to any particular layout of the data or placement in the machine. At runtime, each logical region has one or more *physical instances* assigned to specific memories. It is often useful to have multiple physical instances of a logical region (e.g., to replicate read-only data, or to allow independent reductions that are later combined).

To introduce the programming model, we present a circuit simulation in Section II, illustrating regions, tasks, permissions and coherence properties, the interactions between them, and how these building blocks are assembled into a Legion program. Subsequent sections each describe a contribution in the implementation and evaluation of Legion:

- We define a *software out-of-order processor*, or SOOP, for scheduling tasks with region arguments in a manner analogous to how out-of-order hardware schedulers process instructions with register arguments (Section III). In addition to pipelining the execution of tasks over several stages, our SOOP is distributed across the machine and is also hierarchical to naturally extract nested parallelism (because tasks may recursively spawn subtasks).
- Of central importance is how tasks are assigned (or *mapped*) to processors and how physical instances of logical regions are mapped to specific memory units

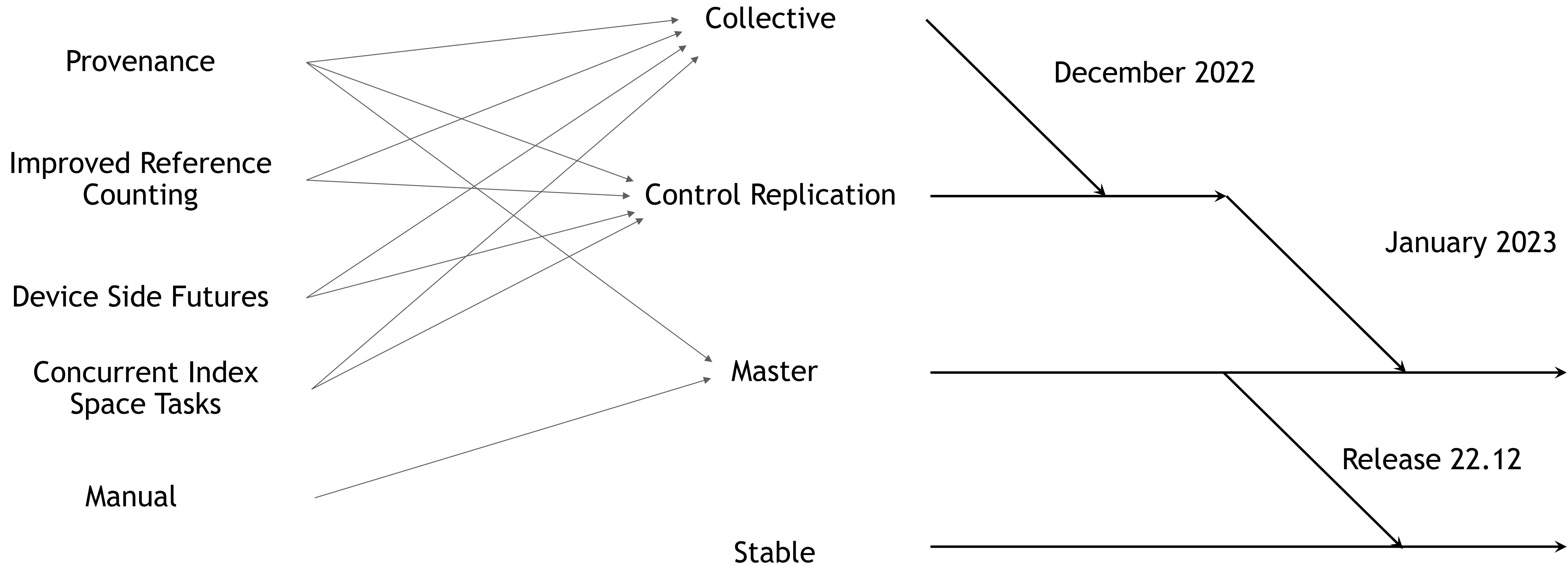
This work was supported by the DARPA UHPC project through a subcontract of NVIDIA (DARPA contract HR0011-10-9-0008), and by the Army High Performance Research Center, through a subcontract from Los Alamos National Laboratory for the DOE Office of Science, Advanced Scientific Computing Research (DE-AC52-06NA25396). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

SC12, November 10-16, 2012, Salt Lake City, Utah, USA

978-1-4673-0806-9/12/ $31.00 © 2012 IEEE

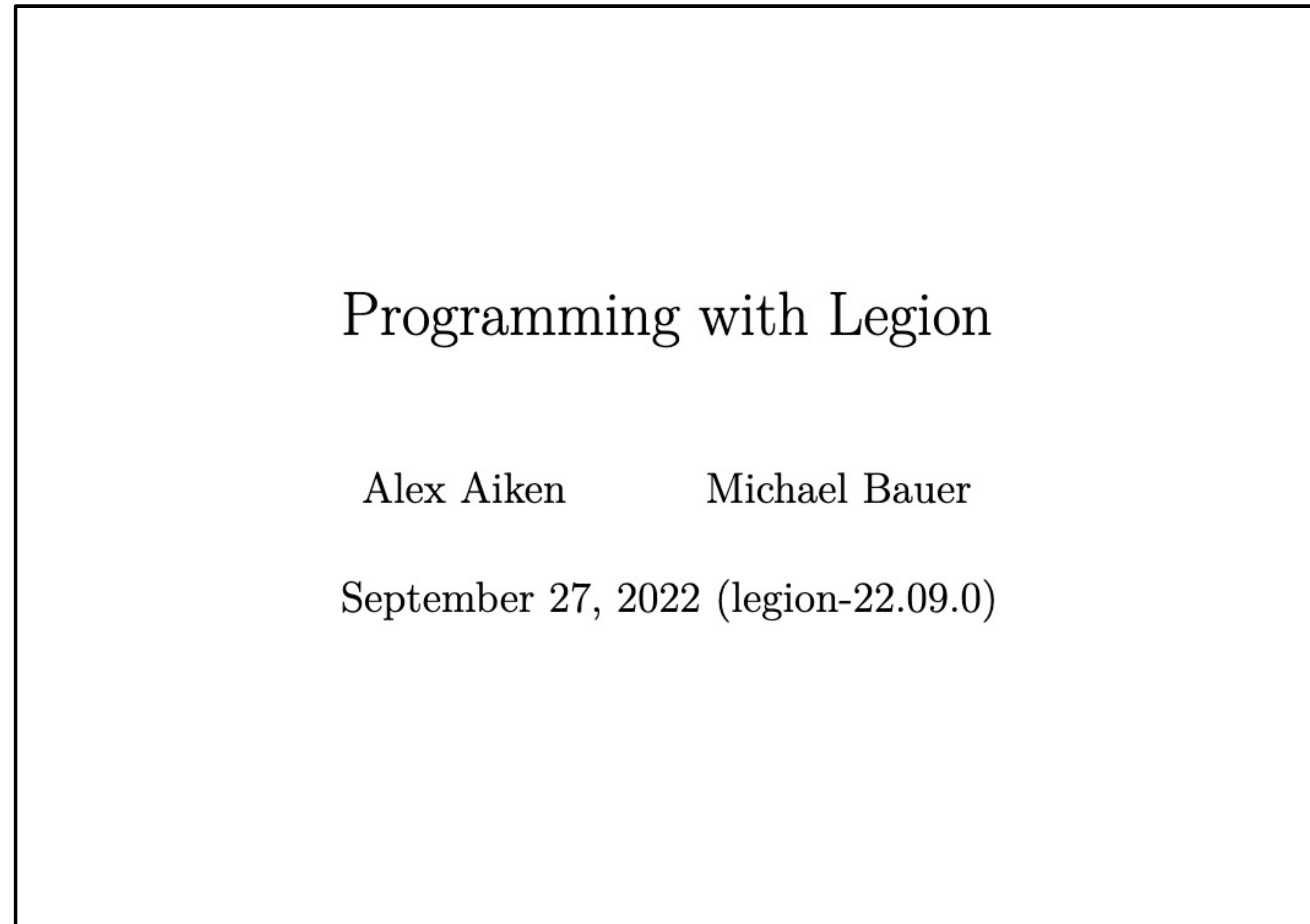
RECENT ROADMAP

What have you done for me lately?



DOCUMENTATION

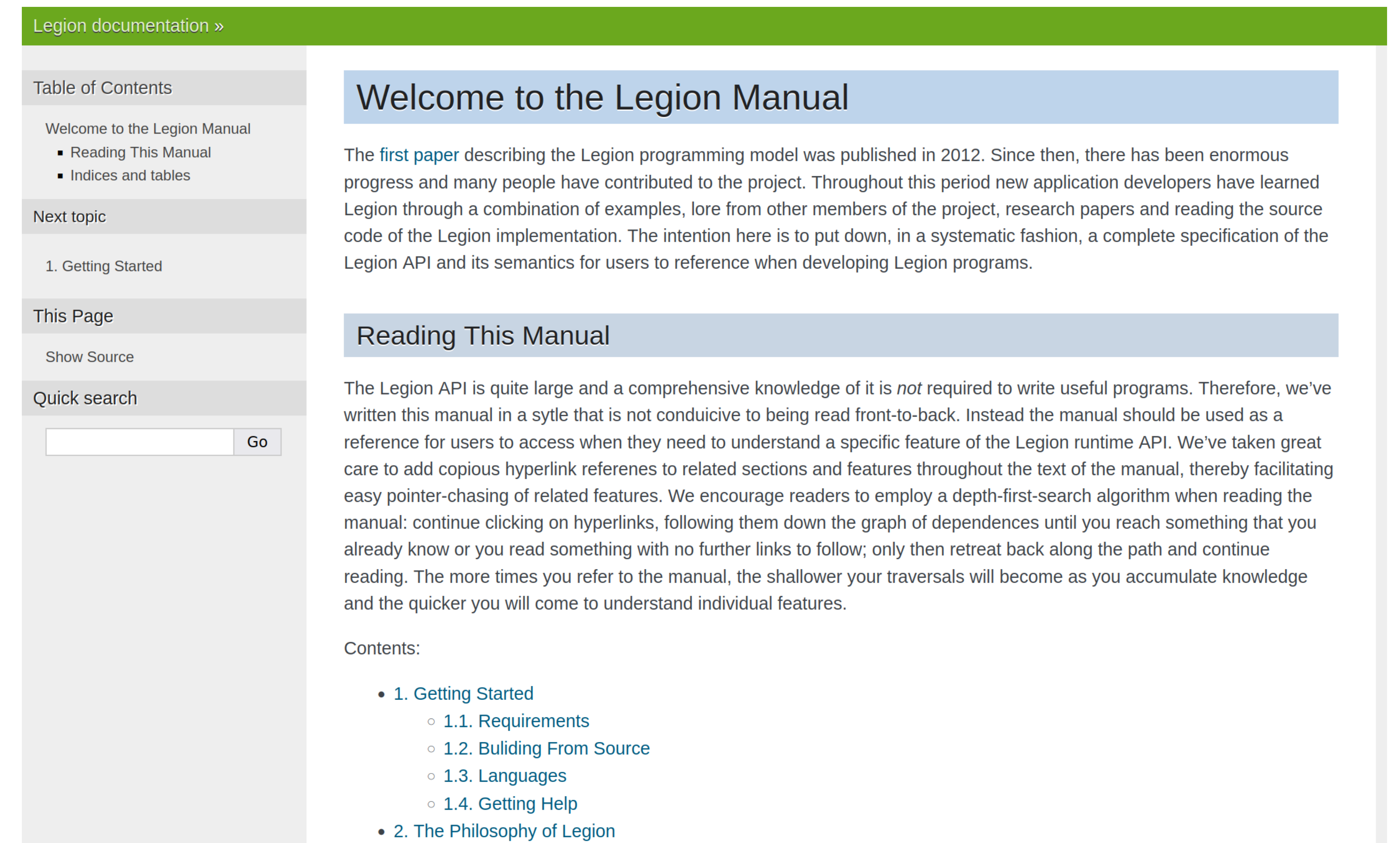
At last



We have a Legion manual!

https://legion.stanford.edu/pdfs/legion-manual.pdf

Idiomatic but incomplete



Sphinx Documentation

https://github.com/StanfordLegion/legion-manual/tree/sphinx

Complete description of all APIs, need help…

PROVENANCE

Where did that come from?

All* Legion APIs now support a provenance string

* Some exceptions may apply (create issue if missing)

Passed through to Legion Prof and Legion Spy

Accessible via the Mappable interface

Eventually used in all error messages

Human- and machine-readable components

Use ‘$’ as delimiter: “human prov $ machine prov”

More in Legion Prof talk

Launchers

```
1507   public:
1508     // If the predicate is set to anything other than
1509     // Predicate::TRUE_PRED, then the application must
1510     // specify a value for the future in the case that
1511     // the predicate resolves to false. UntypedBuffer(NULL,0)
1512     // can be used if the task's return type is void.
1513     Future                                 predicate_false_future;
1514     UntypedBuffer                          predicate_false_result;
1515   public:
1516     // Provenance string for the runtime and tools to use
1517     std::string                            provenance;
1518   public:
1519     // Inform the runtime about any static dependences
1520     // These will be ignored outside of static traces
1521     const std::vector<StaticDependence> *static_dependences;
```

API Calls

```
5362       IndexPartition create_partition_by_restriction(Context ctx,
5363                                   IndexSpace parent,
5364                                   IndexSpace color_space,
5365                                   DomainTransform transform,
5366                                   Domain extent,
5367                                   PartitionKind part_kind = LEGION_COMPUTE_KIND,
5368                                   Color color = LEGION_AUTO_GENERATE_ID,
5369                                   const char *provenance = NULL);
```

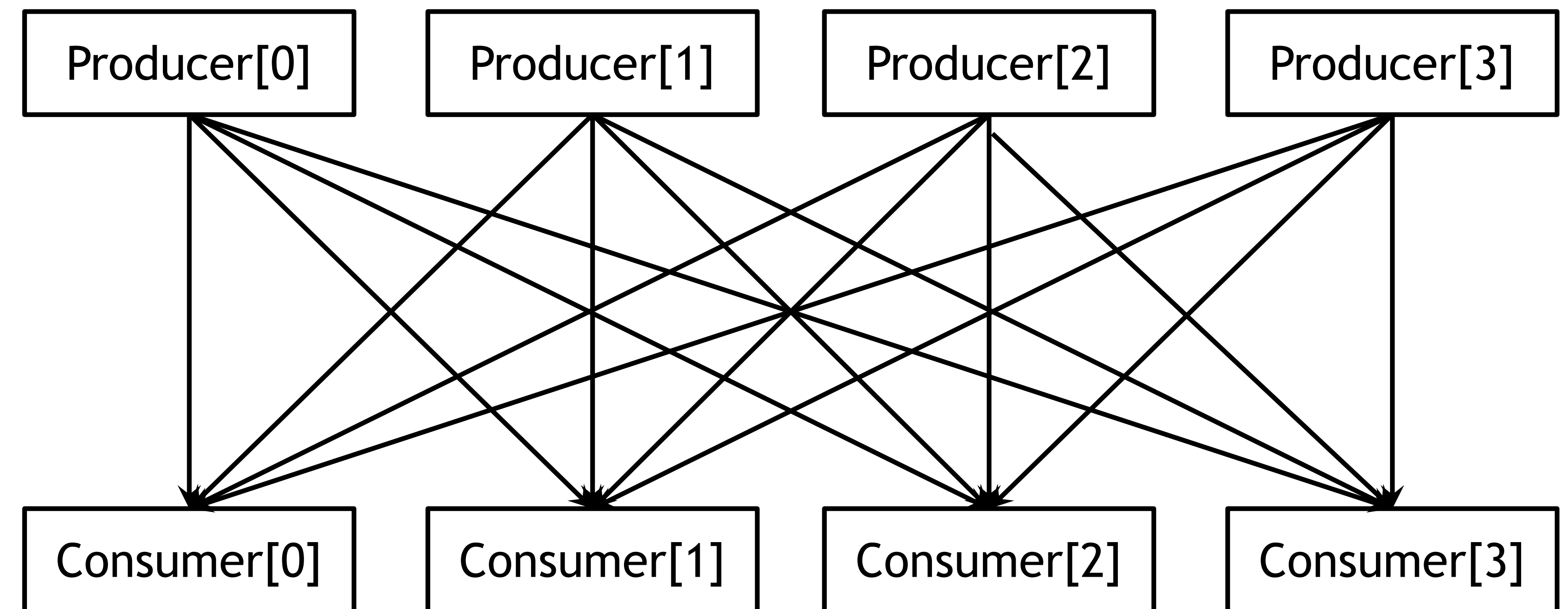
COLLECTIVE VIEWS

Deduplicating analysis

"All-Reduce Program"

Index Task Producer [0-3]: Reduce R.a

Index Task Consumer [0-3]: Read-Only R.a



Every consumer depends on every producer

total # interferences == O(N^2)

Analyzing each task independently doesn't scale

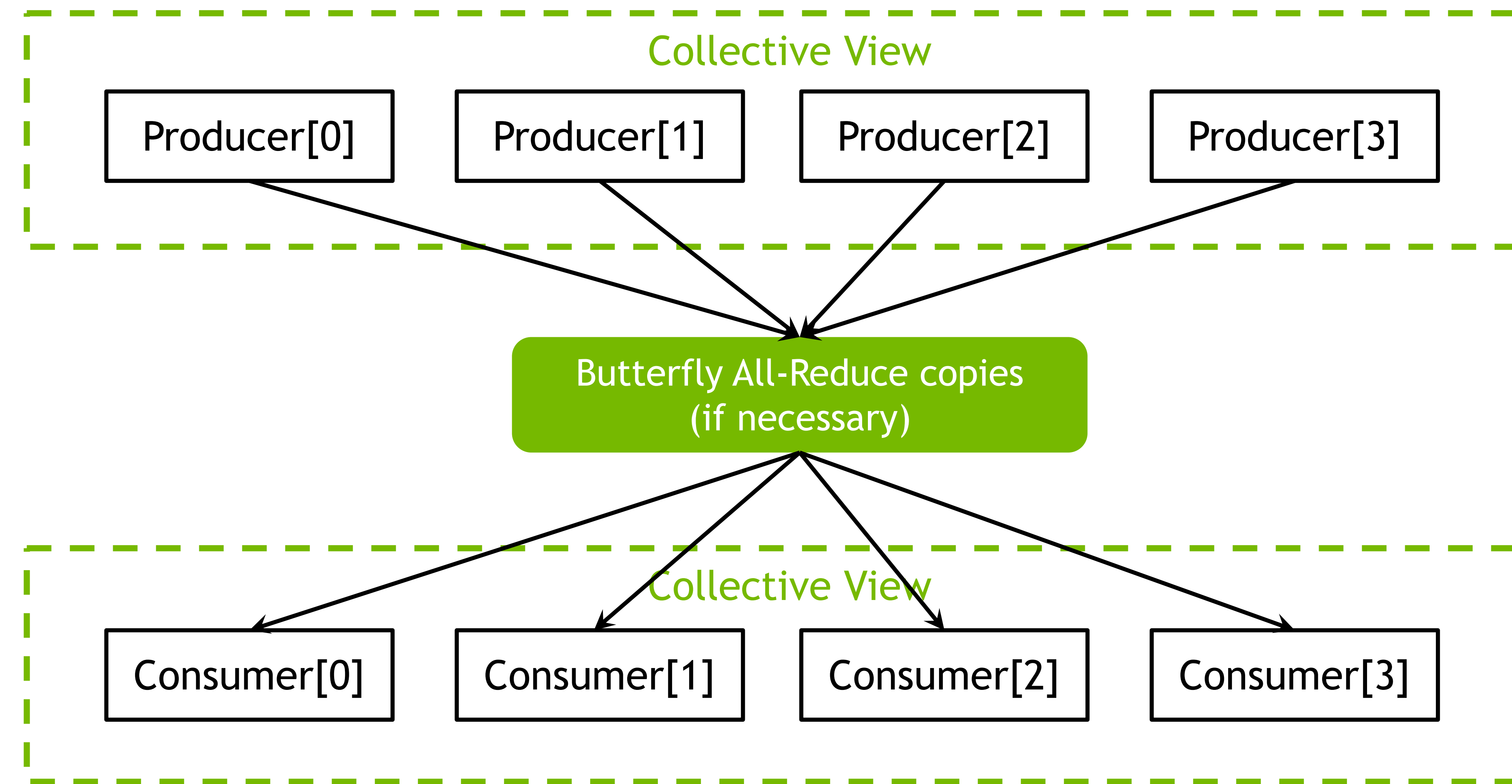
COLLECTIVE VIEWS

An All-Reduce Example

"All-Reduce Program"

Index Task Producer [0-3]: Reduce R.a

Index Task Consumer [0-3]: Read-Only R.a



Foreach task:

Step 1: Mapper (optionally) tells Legion to "look" for collective region usage when mapping tasks

Step 2: Legion performs rendezvous to make collective views for all tasks using the same logical region

Step 3: Legion performs just one dependence analysis for each collective view that it makes

Currently Beta-Testing

Should be mostly stable except for tracing

BROADCAST/REDUCTION

Collective Behavior Comes from Data Usage

"Broadcast Program"

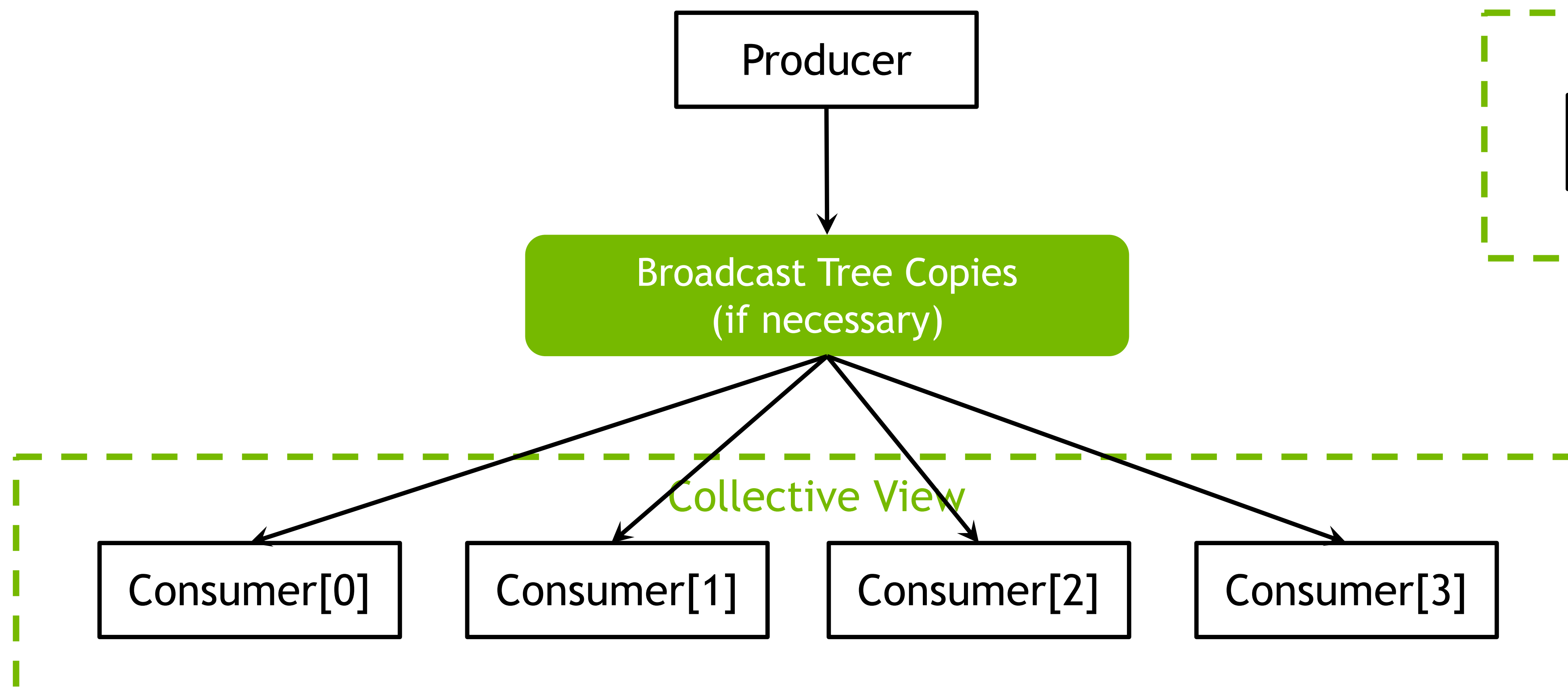
Single Task Producer: Read-Write R.a

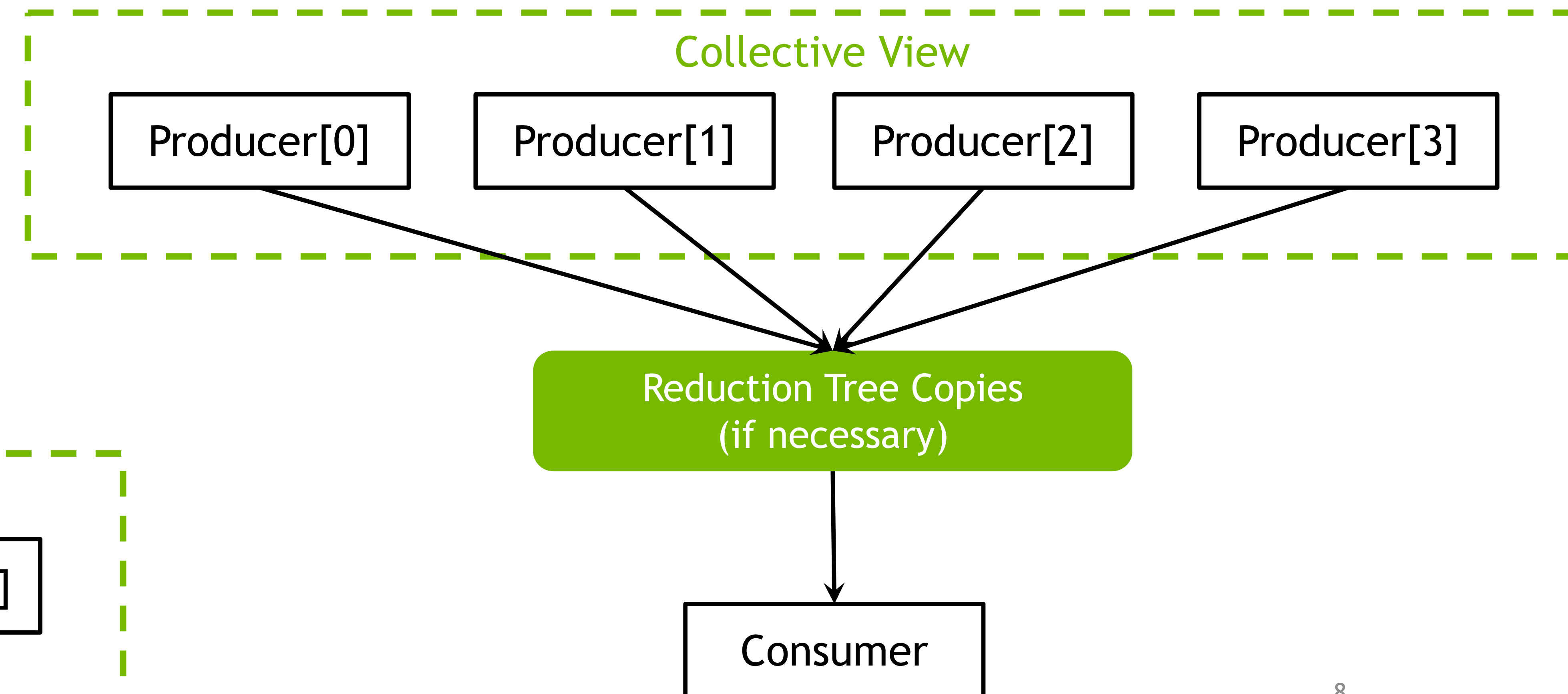
Index Task Consumer [0-3]: Read-Only R.a

"Reduce Program"

Index Task Producer [0-3]: Reduce R.a

Single Task Consumer: Read-Only R.a





(THE END OF) CONTROL REPLICATION

6 years later

Control_Replication_Developers_011717 .PPTX

File Edit View Insert Format Slide Arrange Tools Help

Last edit was on January 16, 2017

Background Layout Theme Transition

LEGION: CONTROL REPLICATION

A REVISIONIST HISTORY OF LEGION S3D

THE PROBLEM

"SHORT TERM" HACK

WHY IS THIS IS A HACK?

FRIENDLY REMINDER

This will be hard

Control replication => deferred distributed parallel program analysis

Any one of these four things by itself is hard

I need as much help as I can get

(THE END OF) CONTROL REPLICATION

6 years later

We're close!

Need collective views

Lots of ops in control replicated tasks implicitly map with collective views

Inline Map, Attach, Detach, Acquire, Release, Fill, etc

Tune equivalence set selection heuristics

Last chance to update the default mapper

Control Replication Merge to Master #765

Open | 5 of 8 tasks | lightsighter opened this issue on Mar 4, 2020 · 3 comments

lightsighter commented on Mar 4, 2020 • edited | Member

See merge request:

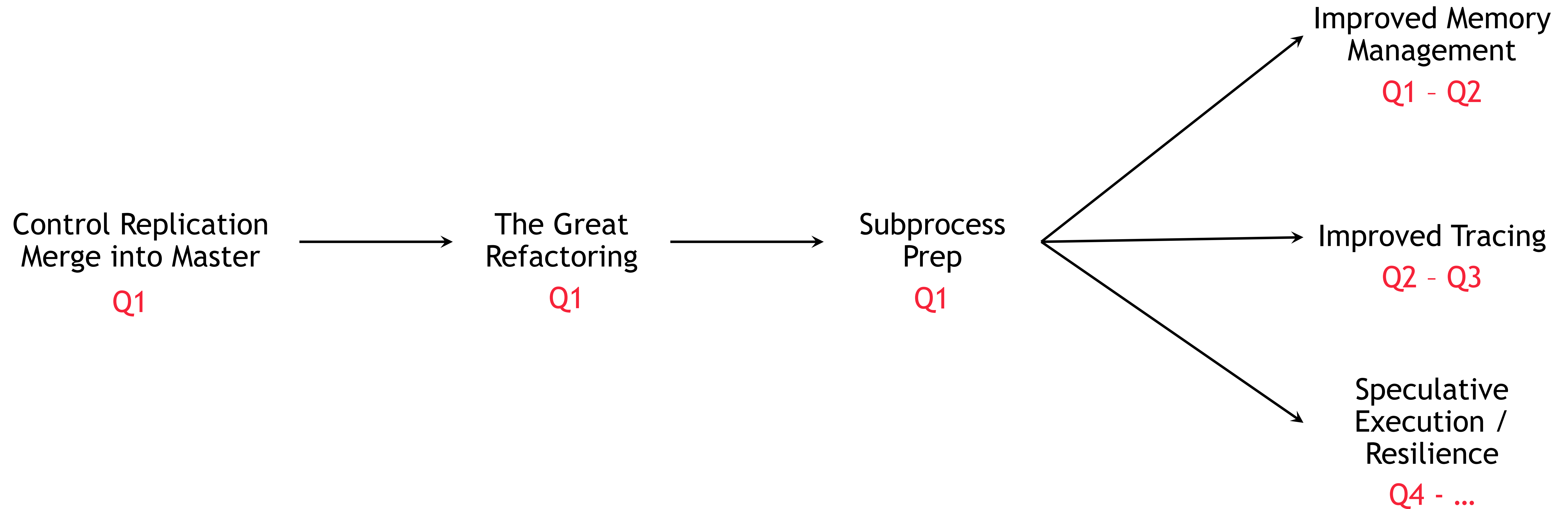
https://gitlab.com/StanfordLegion/legion/-/merge_requests/157

TODO list:

- [ ] Finish refactoring of instances for issue Support for Reduction and Copy Trees #546
- [x] Refactor control replication to use the replicated instances for inline mappings and attach ops
- [x] Teach Legion Spy to validate inline mappings and attach operations in control replication
- [x] Detection of violations of control replication need to be improved
- [x] Verify semantics of attach/detach and inline mappings for control replication
- [ ] Improve default mapper behavior for control replication (tracked in Support for Control Replication in the Default Mapper #896)
- [x] Fix implementation of `premap_task`
- [ ] Ensure we have coverage of non-replicated Regent workloads in CI (@streichler)

NEXT-YEAR ROADMAP

What's next



THE GREAT REFACTORING

Making up for lost time

Legion code needs to be cleaned up (just Legion, not Realm/Regent)

Imposing a clang-based style guide (under development)

All the error messages will be rewritten, e.g. to use provenance among other things

Decouple all error checks from internal debug checks

Clean exits of the runtime even under faults including task stacks

Lots of build parameters will become runtime flags

SUBPROCESS PREP

First steps towards strong task isolation

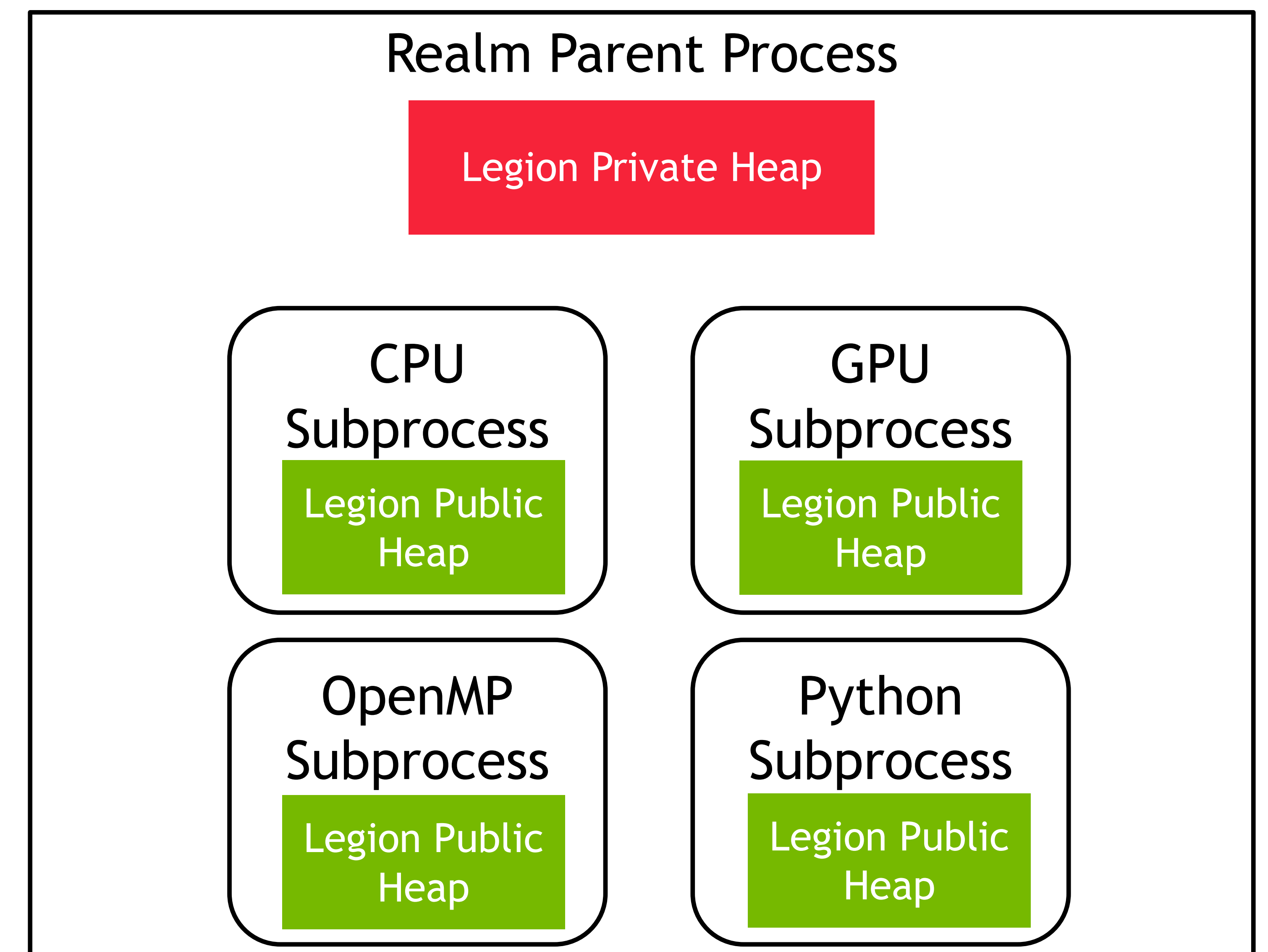
Goal: each Realm processor executes task in its own subprocess

Why? Isolation: see global variables in Python, OpenMP, etc.

Potentially also isolate regions for each task using mprotect

Catch: still need to map parts of Legion and Realm runtimes into subprocess for task execution

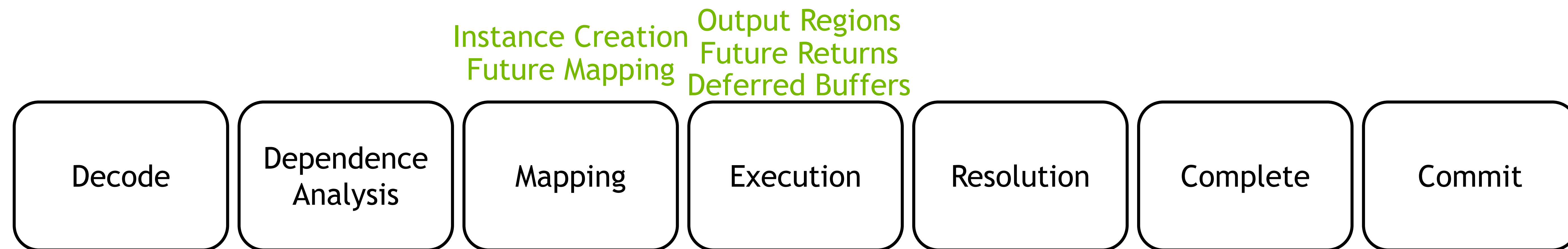
Every single Legion memory allocation needs to be put in the right heap



IMPROVED MEMORY MANAGEMENT

Probably our most pressing need

When are instances allocated in the pipeline:



Today we use two pools: deferred allocations in mapping and eager allocations during execution

Better: one pool with allocation shifting back and forth on demand

Requirement: need upper bound on execution allocations required by each task

Open problem: how to deal with instance allocation failures in the mapper

IMPROVED TRACING

Legion as a JIT-ing interpreter

Tracing is vital to our success: 5-20X faster
Must be the common case for all Legion execution

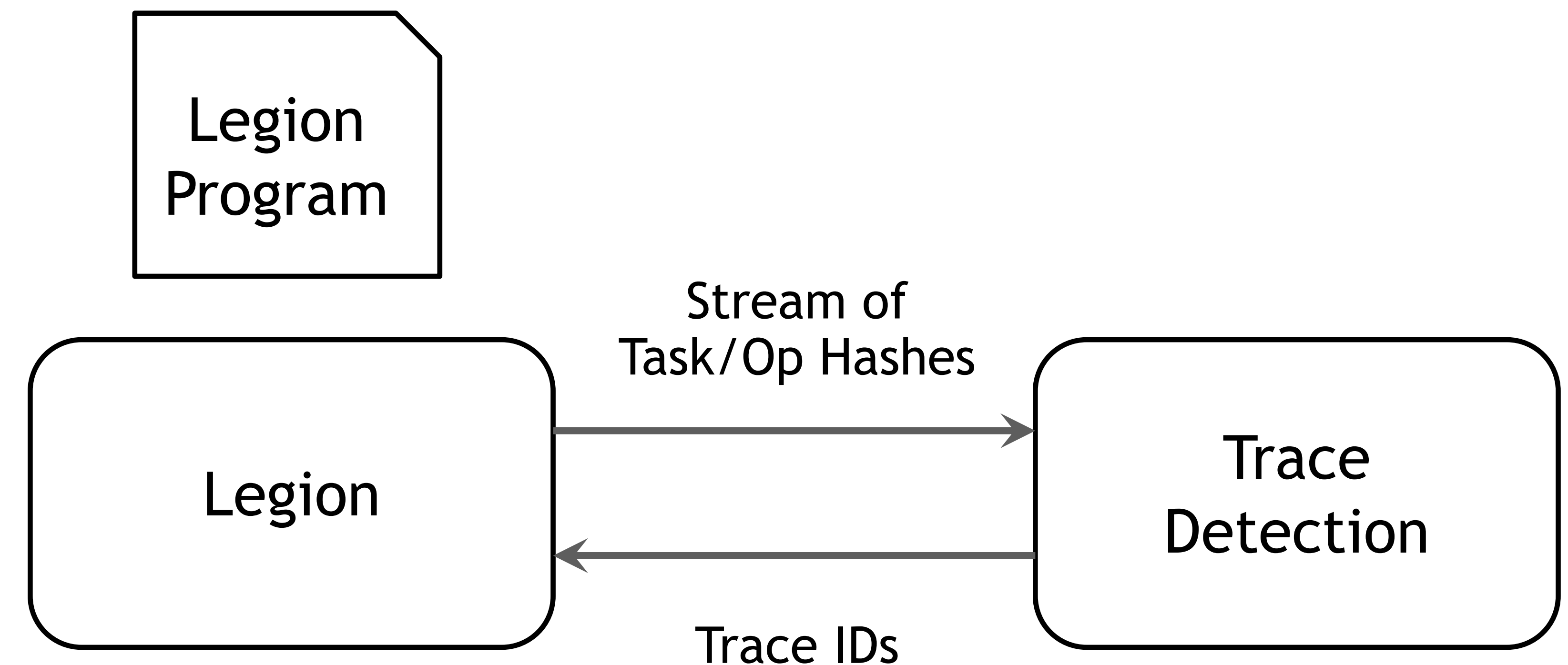
Non-idempotent traces

Checks for safe tracing

Automatic inference of traces
(longest common subsequence)

Lowering to Realm graphs

Open problem: memory reuse inside of traces?



SPECULATIVE EXECUTION / RESILIENCE

Two side of the same coin

Today we have predicated execution

Starting to see need for speculative execution
(LANL, Legate, output regions)

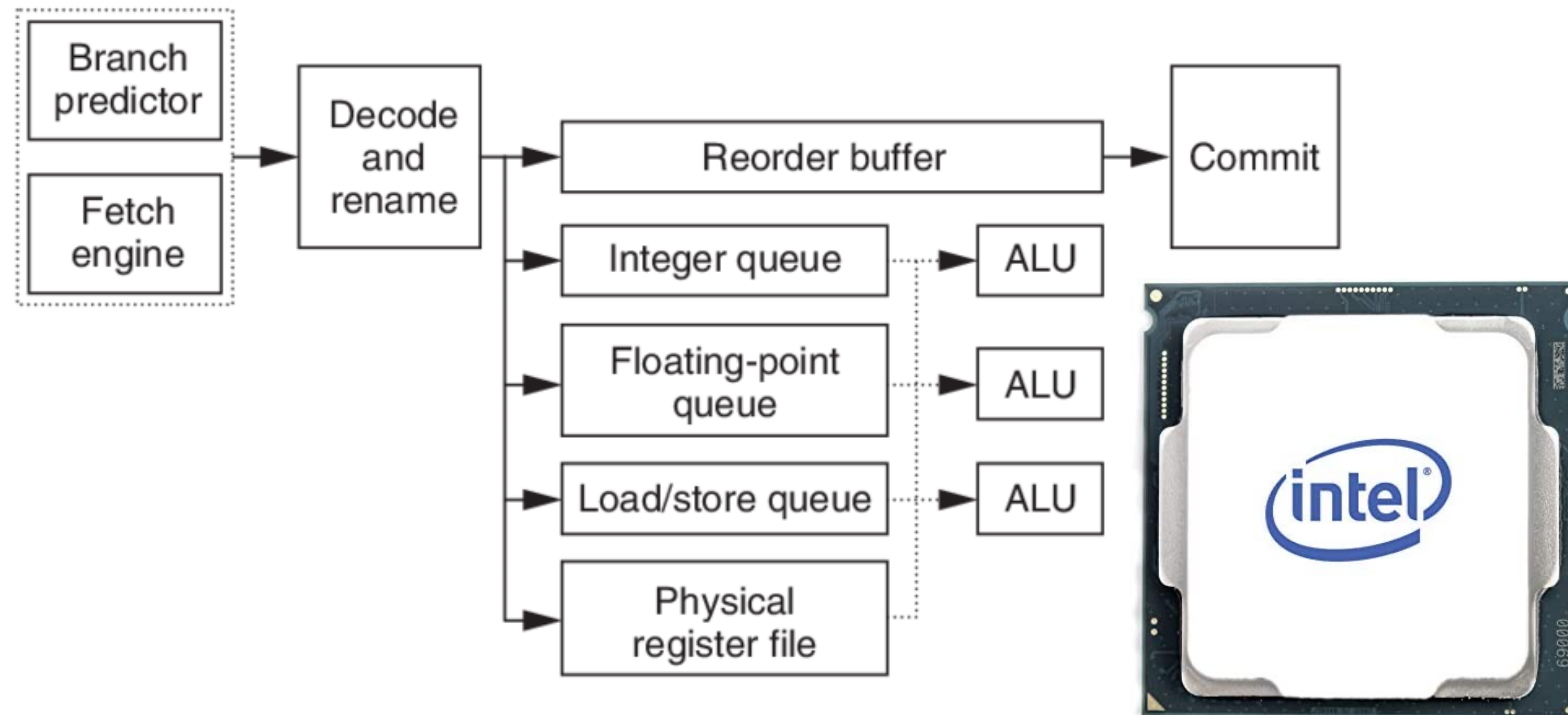
Explicit API calls to “branch” / “checkpoint” in Legion

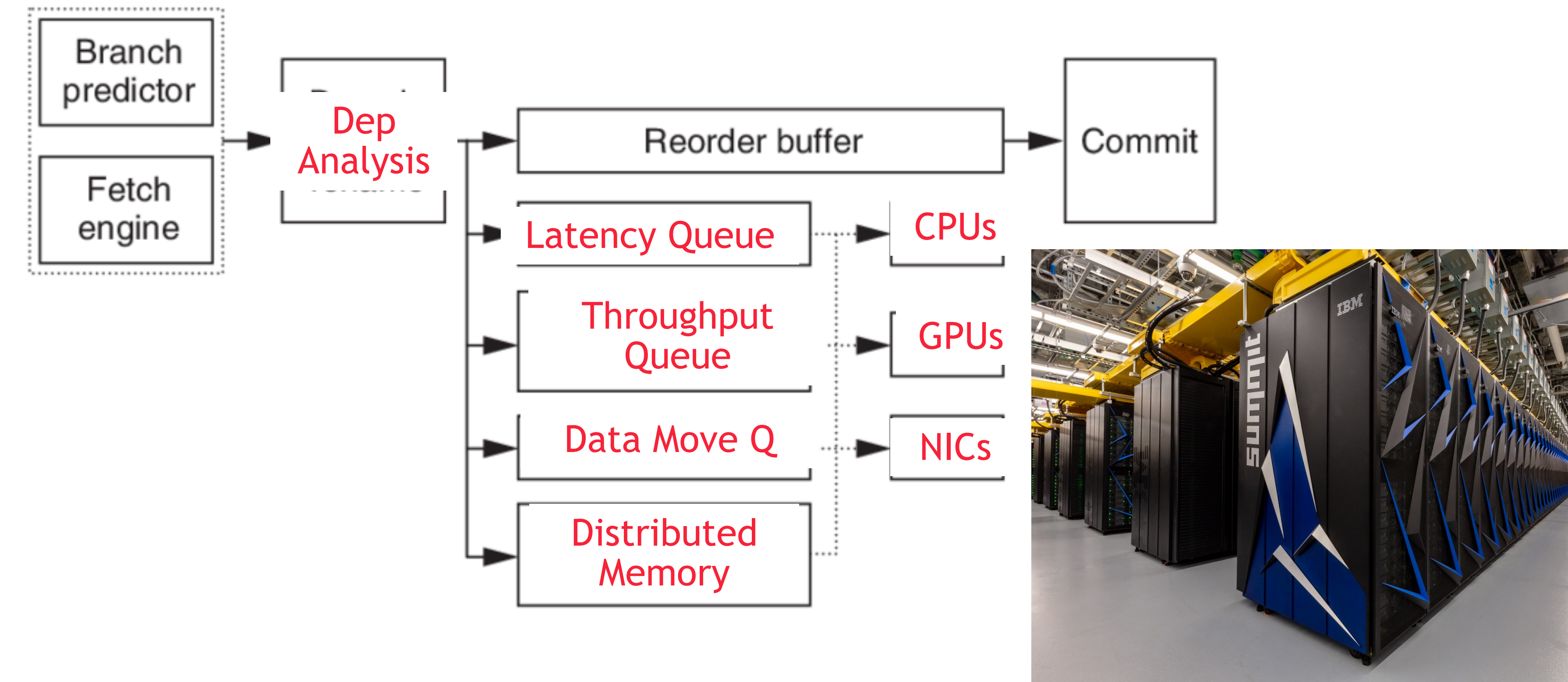
Runtime can recover back if misspeculate/fault

Open problem: how to say when in-place updates allowed?

STRATEGY

Advantage 1: implicit parallelism with control replication



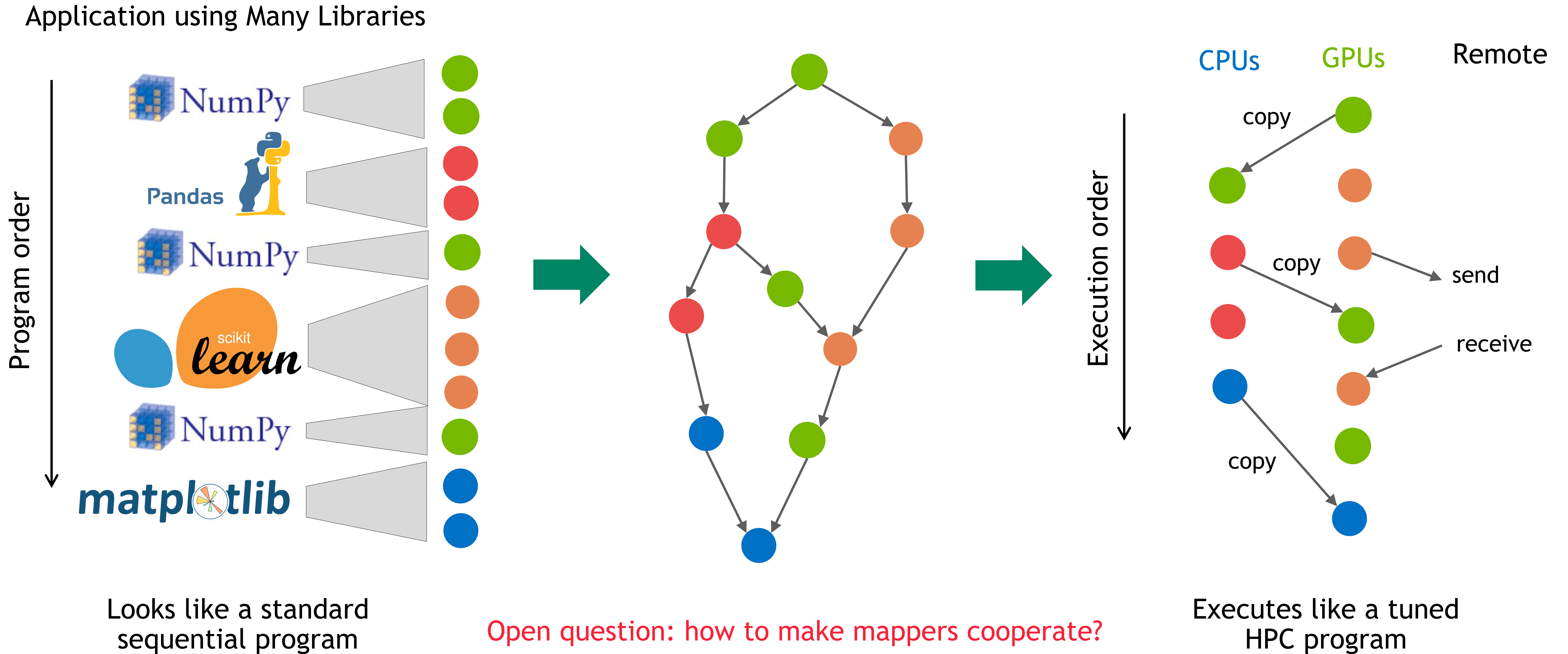


Push this analogy hard: it was successful for decades, no reason it won't be successful at larger scales in software

Control replication makes this approach scalable, everyone else is years behind on us on this

STRATEGY

Advantage 2: software composability



CONCLUSION

A bright future

We’re past the point of ‘max implementation complexity’

No new programming model features, just better implementations of existing features

We have crucial architectural advantages over our competitors that we need to exploit

Legion

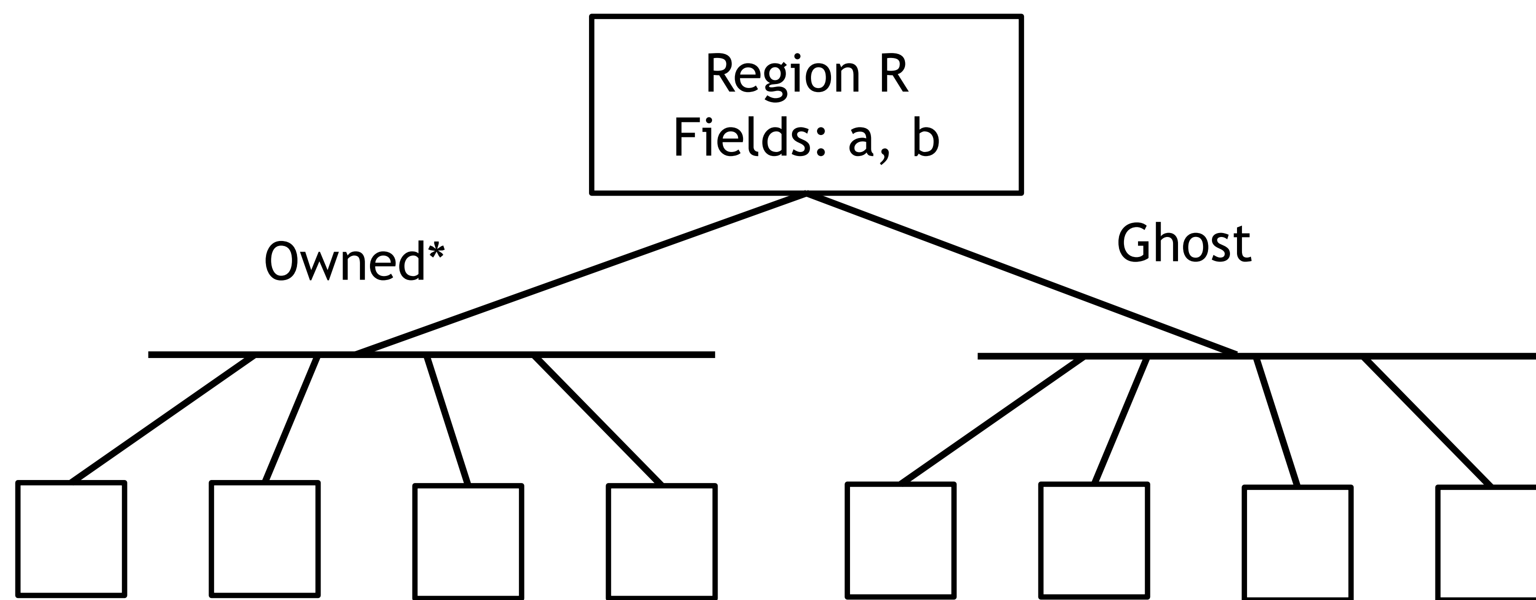
“SPARSE INTERFERENCE” TASKS

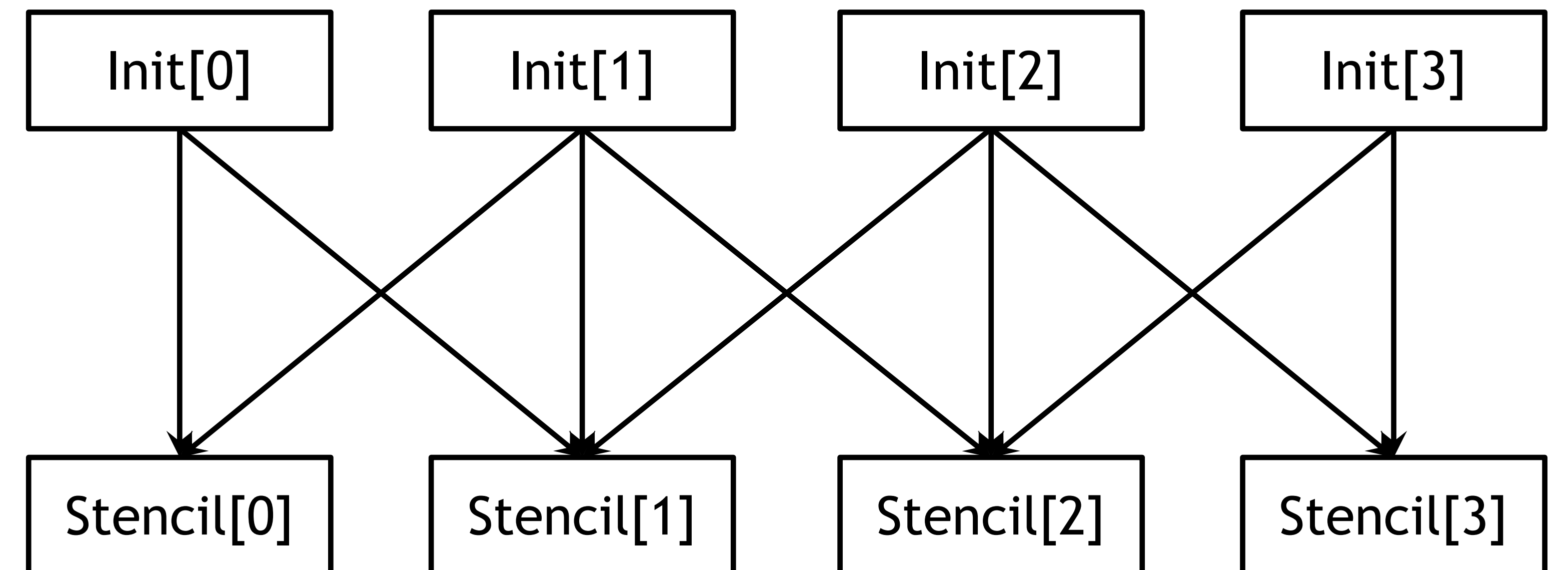
Common Case

”Stencil Program”

Index Task Init [0-3]: Read-Write Owned[i].a

Index Task Stencil [0-3]: Read-Write Owned[i].b
Read-Only Ghost[i].a





Dependence analysis for each task performed independently

As N gets larger, average number of interferences with prior tasks is bounded

total # interferences << O(N^2)

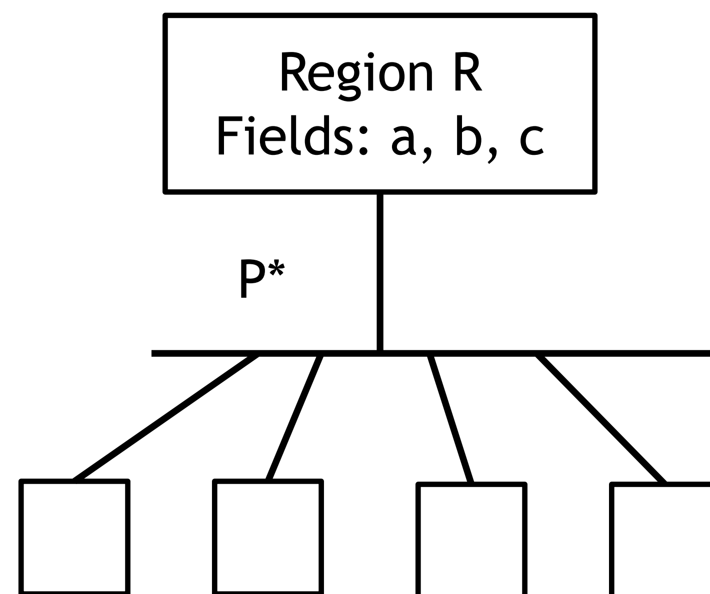
REGION MATCHING

Not All Tasks Need to Use the Same Region

"GEMM Program"

Index Task Init [M,N]: Read-Write P[m,n].a,b,c

Index Task GEMM [M,N,K]: Reduce P[f1(m,n,k)].c
Read-Only P[f2(m,n,k)].a
Read-Only P[f3(m,n,k)].b



Projection functions can map subsets of point tasks to the same region

Runtime will discover which subsets of points in the index launch share common regions and can be analyzed collectively

All points in the index launch must still participate in the match!

COMPOSABILITY

No Communicators Required

"All-Reduce Program"

M != N

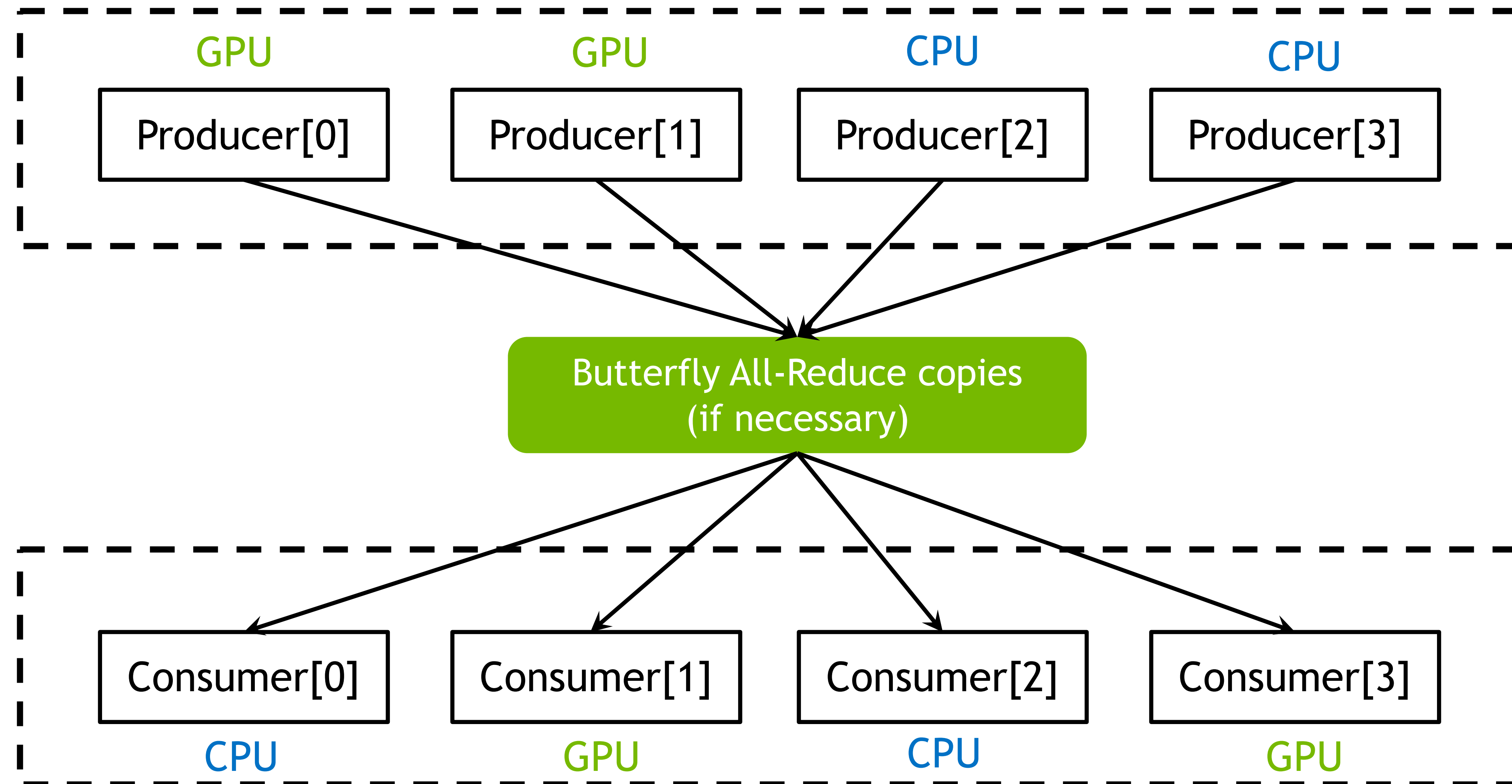
M < N, M == N, M > N
Get collective behavior in all cases

Library A — Index Task Producer [0-M]: Reduce R.a

Library B — Index Task Consumer [0-N]: Read-Only R.a

Libraries don't need to know about each other's collective index tasks to get collective behavior

Works regardless of where tasks map, can mix and match CPUs and GPUs arbitrarily



Collective dependences and data movement are just part of the Realm event graph, automatically overlap with other independent computation or data movement, no synchronization required

PERFORMANCE CONSIDERATIONS

Tradeoffs

Rendezvous and creation are both O(log N) in the number of point tasks, don’t turn it on unless you think there actually is collective regions in the index launch

You’ll probably need a little bit of task parallelism to hide this latency at scale

Collective rendezvous memoized under tracing (more incentive to get that working with Legate)

Try to re-use the same instances for collective views, the runtime will de-duplicate collective views and not need to recreate them

Runtime builds hierarchical collective copies if multiple instances on the same node (prefer process-per-node)

Foreach task:

Step 1: Mapper (optionally) tells Legion to “look” for collective region usage when mapping tasks

Step 2: Legion performs rendezvous to make collective views for all tasks using the same logical region

Step 3: Legion performs just one dependence analysis for each collective view that it makes